Simulations parallèles de Monte Carlo appliquées à la Physique des Hautes Energies pour plates-formes manycore et multicore : mise au point, optimisation, reproductibilité

Pierre Schweitzer

▶ **To cite this version:**

Pierre Schweitzer. Simulations parallèles de Monte Carlo appliquées à la Physique des Hautes Energies pour plates-formes manycore et multicore : mise au point, optimisation, reproductibilité. Autre [cs.OH]. Université Blaise Pascal - Clermont-Ferrand II, 2015. Français. NNT : 2015CLF22605 . tel-01386468

HAL Id: tel-01386468

https://theses.hal.science/tel-01386468

Submitted on 24 Oct 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Blaise Pascal

École Doctorale des Sciences pour l'Ingénieur
N° d'ordre : 2605 - EDSPIC : 711

Thèse

pour l'obtention du grade de

DOCTEUR D'UNIVERSITÉ

Discipline : Informatique

Présentée par

Pierre SCHWEITZER

le 19/10/2015

Simulations parallèles de Monte Carlo appliquées à la Physique des Hautes Énergies pour plates-formes `manycore` et `multicore` : mise au point, optimisation, reproductibilité

Composition du jury :

Rapporteurs :	Stéphane VIALLE, Professeur, Supélec Metz
	Joël FALCOU, Maître de conférences HDR, Université Paris-Sud
Examinateurs :	David ROUSSEAU, Directeur de recherche, LAL
	Christophe HAEN, Ingénieur de recherche, CERN
Directeur de thèse :	David R.C. HILL, Professeur, Université Blaise Pascal
Codirectrice de thèse :	Cristina CÂRLOGANU, Chargée de recherche, LPC
Invité :	Claude MAZEL, Maître de conférences, Université Blaise Pascal

Résumé

Lors de cette thèse, nous nous sommes focalisés sur le calcul à haute performance, dans le domaine très précis des simulations de Monte Carlo appliquées à la physique des hautes énergies, et plus particulièrement, aux simulations pour la propagation de particules dans un milieu. Les simulations de Monte Carlo sont des simulations particulièrement consommatrices en ressources, temps de calcul, capacité mémoire.

Dans le cas précis sur lequel nous nous sommes penchés, la première simulation de Monte Carlo existante prenait plus de temps à simuler le phénomène physique que le phénomène lui-même n'en prenait pour se dérouler dans les conditions expérimentales. Cela posait donc un sévère problème de performance. L'objectif technique minimal était d'avoir une simulation prenant autant de temps que le phénomène réel observé, l'objectif maximal était d'avoir une simulation bien plus rapide. En effet, ces simulations sont importantes pour vérifier la bonne compréhension de ce qui est observé dans les conditions expérimentales. Plus nous disposons d'échantillons statistiques simulés, meilleurs sont les résultats. Cet état initial des simulations ouvrait donc de nombreuses perspectives d'un point de vue optimisation et calcul à haute performance. Par ailleurs, dans notre cas, le gain de performance étant proprement inutile s'il n'est pas accompagné d'une reproductibilité des résultats, la reproductibilité numérique de la simulation est de ce fait un aspect que nous devons prendre en compte.

C'est ainsi que dans le cadre de cette thèse, après un état de l'art sur le profilage, l'optimisation et la reproductibilité, nous avons proposé plusieurs stratégies visant à obtenir plus de performances pour nos simulations. Dans tous les cas, les optimisations proposées étaient précédées d'un profilage. On n'optimise jamais sans avoir profilé. Par la suite, nous nous intéressés à la création d'un profileur parallèle en programmation orientée aspect pour nos besoins très spécifiques, enfin, nous avons considéré la problématique de nos simulations sous un angle nouveau : plutôt que d'optimiser une simulation existante, nous avons proposé des méthodes permettant d'en créer une nouvelle, très spécifique à notre domaine, qui soit d'emblée reproductible, statistiquement correcte et qui puisse passer à l'échelle. Dans toutes les propositions, de façon transverse, nous nous sommes intéressés aux architectures *multicore* et *manycore* d'Intel pour évaluer les performances à travers une architecture orientée serveur et une architecture orientée calcul à haute performance.

Ainsi, grâce à la mise en application de nos propositions, nous avons pu optimiser une des simulations de Monte Carlo, nous permettant d'obtenir un gain de performance de l'ordre de 400X, une fois optimisée et parallélisée sur un nœud de calcul avec 32 cœurs physiques. De même, nous avons pu proposer l'implémentation d'un profileur, programmé à l'aide d'aspects et capable de gérer le parallélisme à la fois de la machine sur laquelle il est exécuté mais aussi de l'application qu'il profile. De plus, parce qu'il emploi les aspects, il est portable et n'est pas fixé à une architecture matérielle en particulier. Enfin, nous avons implémenté la simulation prévue pour être reproductible, performante et ayant des résultats statistiquement viables. Nous avons pu constater que ces objectifs étaient atteints quelle que soit l'architecture cible pour l'exécution. Cela nous a permis de valider notamment notre méthode de vérification de la reproductibilité numérique d'une simulation.

Abstract

During this thesis, we focused on High Performance Computing, specifically on Monte Carlo simulations applied to High Energy Physics. We worked on simulations dedicated to the propagation of particles through matter. Monte Carlo simulations require significant CPU time and memory footprint.

Our first Monte Carlo simulation was taking more time to simulate the physical phenomenon than the said phenomenon required to happen in the experimental conditions. It raised a real performance issue. The minimal technical aim of the thesis was to have a simulation requiring as much time as the real observed phenomenon. Our maximal target was to have a much faster simulation. Indeed, these simulations are critical to asses our correct understanding of what is observed during experimentation. The more we have simulated statistics samples, the better are our results. This initial state of our simulation was allowing numerous perspectives regarding optimisation, and high performance computing. Furthermore, in our case, increasing the performance of the simulation was pointless if it was at the cost of losing results reproducibility. The numerical reproducibility of the simulation was then an aspect we had to take into account.

In this manuscript, after a state of the art about profiling, optimisation and reproducibility, we proposed several strategies to gain more performance in our simulations. In each case, all the proposed optimisations followed a profiling step. One never optimises without having profiled first. Then, we looked at the design of a parallel profiler using aspect-oriented programming for our specific needs. Finally, we took a new look at the issues raised by our Monte Carlo simulations: instead of optimising existing simulations, we proposed methods for developing a new simulation from scratch, having in mind it is for High Performance Computing and it has to be statistically sound, reproducible and scalable. In all our proposals, we looked at both multicore and manycore architectures from Intel to benchmark the performance on server-oriented architecture and High Performance Computing oriented architecture.

Through the implementation of our proposals, we were able to optimise one of the Monte Carlo simulations, permitting us to achieve a 400X speedup, once optimised and parallelised on a computing node with 32 physical cores. We were also able to implement a profiler with aspects, able to deal with the parallelism of its computer and of the application it profiles. Moreover, because it relies on aspects, it is portable and not tied to any specific architecture. Finally, we implemented the simulation designed to be reproducible, scalable and to have statistically sound results. We observed that these goals could be achieved, whatever the target architecture for execution. This enabled us to assess our method for validating the numerical reproducibility of a simulation.

Remerciements

Mes premiers remerciements vont aux personnes sans qui je n'aurais jamais pu réaliser cette thèse, à commencer par mes deux encadrants David Hill et Cristina Cârloganu. Je tiens également à remercier très chaleureusement Claude Mazel qui m'a beaucoup encadré, aidé, relu, dépanné, sauvé durant l'intégralité de cette thèse. Ses conseils et sa réflexion m'ont été d'une grande aide. Il m'est nécessaire de remercier également Valentin Niess, pour sa patience et sa grande pédagogie quand il fallait m'expliquer la physique derrière ToMuVol, m'expliquer pourquoi nos simulations fonctionnaient d'une façon et pas d'un autre. Je lui dois beaucoup. Un grand merci également à Jonathan Passerat-Palmbach pour son aide et son soutien dès le début de la thèse (et même avant !).

Je tiens à remercier également Samuel Béné pour son aide sur les aspects physiques liés à la thèse, ainsi que pour deux figures dans ce manuscrit.

Je me dois de remercier également, chaleureusement, les personnels administratifs et techniques qui m'ont aidé et soutenu durant ma thèse : Séverine Miginiac, Aurélie Bavent, Béatrice Bourdieu, Socheata Sean et Nicolas Champeil. Sans eux, ma thèse aurait été bien plus monotone, et administrativement beaucoup plus contraignante.

Merci également à toutes les personnes des équipes PCSV et IdGC du LPC chez qui je me suis « un peu » invité au fil de ma thèse et qui m'ont accueilli très gentiment : Lydia Maigne, Vincent Breton, Henri Payno, Sébastien Cipière, Silvia Gervois, Géraldine Fettahi, David Saramia.

Merci aussi à toutes les personnes que j'ai pu côtoyer durant l'intégralité de ma thèse dans mes différents bureaux, et autres laboratoires et qui m'ont apporté bonne humeur : Nathalie Klement, Simon Nicolas, Romain Lardy, Wajdi Dhifli, Faouzi Jaziri, Luc Touraille, Pierre Marin, Nathanael Lampe, Bogdan Vulupescu, … Plus tous ceux que j'oublie ; rembobiner sur trois ans, ce n'est pas le plus évident !

Un merci particulier à Christophe Duhamel, qui a cru en moi dès le début.

Et enfin, les deux derniers mercis pour Florence Guillebert, pour son aimable relecture du manuscrit et Susan Arbon-Leahy pour ses différentes relectures des articles (et résumés) écrit en Anglais.

Table des matières

Table des figures

Table des tableaux

Tableau des codes

Introduction

1 – Contexte

Cette thèse et l'intégralité des travaux qui y ont été réalisés l'ont été dans le cadre de l'expérience ToMuVol[1]. L'expérience ToMuVol, rattachée au laboratoire d'excellence (LabEx) ClerVolc[2] vise à permettre de mettre au point des outils tant informatiques que matériels permettant de réaliser la tomographie de grands édifices. Le LabEx Clervolc, Centre Clermontois de Recherche sur le Volcanisme vise à étudier et enseigner plusieurs domaines en rapport avec le volcanisme, y compris les dangers et les risques liés au volcanisme.

La tomographie consiste à réaliser, tranche par tranche, une représentation en trois dimensions d'un édifice. On retrouve cet usage dans la vie courante, notamment en imagerie médicale avec les scanners : la tomodensitométrie. Celle-ci permet d'obtenir une image d'un organe du corps pour analyse médicale. On place le patient au centre du scanner qui sera constitué d'un anneau rotatif. D'un côté, il y a aura des dispositifs d'émission de rayon X (des photons), et diamétralement opposé, des détecteurs. Des rayons X vont alors être émis, vont traverser le corps de la personne, puis iront taper les détecteurs qui mesureront le flux restant de rayons X. Dès lors, le flux initial et le flux final étant connus, il est possible de calculer la transmittance de la cible : c'est-à-dire le rapport entre le flux émis et le flux mesuré. Chaque rotation de l'anneau correspond à une prise de vue et grâce aux données enregistrées pour chaque prise de vue une reconstruction (mathématique) permet de récupérer la représentation du corps en trois dimensions. En lien avec le Laboratoire de Physiques Corpusculaire de Clermont-Ferrand (LPC), le Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes (LIMOS) a une longue expérience autour des simulations parallèles de Monte Carlo pour la tomographie en trois dimensions et la médecine nucléaire (El Bitar et al., 2006; Lazaro, El Bitar, Breton, Hill, & Buvat, 2005; Reuillon, Hill, El Bitar, & Breton, 2008).

Dans le cadre de ToMuVol, nous nous intéressons à des édifices bien plus grands que des organes. En effet, ToMuVol vise à permettre d'imager l'intérieur de volcans, de pyramides, de cœurs de centrales nucléaires, ou encore de hauts fourneaux. Ces cibles ont été sélectionnées parce qu'elles présentent un intérêt de surveillance (qu'est-ce qu'il s'y passe ?) ou encore de connaissance (comment est-ce à l'intérieur ?). Dans le cas de cette tomographie précisément, l'électron ou le

[1] http://www.tomuvol.fr/
[2] http://clervolc.univ-bpclermont.fr/

photon ne sont plus des particules appropriées pour réaliser la tomographie puisque pas assez pénétrantes.

C'est pourquoi ToMuVol s'appuie sur une particule similaire à l'électron : le muon. Celui est présent à l'état naturel et présente des propriétés similaires à l'électron, si ce n'est qu'il est 200 fois plus lourd. C'est de cette particule que l'expérience ToMuVol tire son nom : Tomographie Muonique des Volcans. Le principe est simple, et similaire à ce qui se fait en imagerie médicale : le nombre et la direction des muons ayant traversé la cible qui nous intéresse sont mesurés à l'aide d'un détecteur, ainsi que ceux étant passés à côté de la cible. La mesure du nombre de muons étant passés à côté de la cible permet de normaliser le nombre des muons traversant celle-ci dans chaque direction et d'en déduire la carte de transmittance de la cible. Il s'agit donc d'une image radiographique, bi-dimensionnelle. Des algorithmes prenant en compte les interactions des muons avec la matière permettent d'obtenir à partir de la carte de transmittance la carte bi-dimensionnelle de densité de l'édifice. Bi-dimensionnelle puisque comme dans le cas d'une radiographie à rayons X la mesure fournit l'intégrale de la densité au long de chaque direction sondée. De là, si plusieurs angles de vue de la cible sont disponibles en changeant par exemple l'emplacement du détecteur autour de la cible, celle-ci peut être reconstruite comme pour l'imagerie médicale, en combinant par un algorithme approprié (par exemple, la transformée de Radon (Radon, 1917, 1986)) les différentes tranches mesurées dans une image 3D.

En parallèle de la prise de données réelles, effectuée par le détecteur, il est nécessaire de réaliser des simulations reproduisant le phénomène naturel observé. Ces simulations sont nécessaires parce qu'il faut s'assurer de la bonne compréhension de ce que nous observons avec le détecteur. Et, si nous ne comprenons pas ce que nous observons, itérer sur les simulations jusqu'à obtenir un résultat similaire permettra de mieux comprendre ce que nous observons. En effet, le premier gain de l'activité de modélisation n'est pas de pouvoir réaliser des « prédictions », mais bien d'améliorer notre compréhension des phénomènes étudiés. A titre d'exemple, les simulations pourront servir à comprendre et à modéliser le bruit de fond que nous avons dans les observations réelles. Par ailleurs, cela signifie qu'il faudra plusieurs itérations sur un modèle théorique pour tenter d'approcher les mesures physiques sur le véritable modèle.

Parce que la propagation des muons à travers la cible, ainsi que leur génération en haute atmosphère sont des processus stochastiques, les simulations qui sont implémentées sont des simulations de Monte Carlo. Il existe des modèles analytiques permettant de s'affranchir des processus stochastiques, cependant ceux-ci s'appuyant sur des moyennes, ils perdent en précision et ne permettent pas de travailler finement sur les grands espaces simulés. De ce fait, des simulations

de Monte Carlo sont donc utilisées pour l'expérience ToMuVol, en plus des programmes plus rapides basés sur des calculs analytiques.

Il est de « notoriété publique » pour les modélisateurs, que les simulations de Monte Carlo ne sont pas les simulations les plus rapides, ni les plus légères. Il est moins connu, mais pourtant tout aussi critique qu'il est nécessaire d'implémenter correctement les aspects stochastiques si l'on veut des résultats statistiquement valables pour des simulations parallèles massives. Ainsi, on se retrouve avec plusieurs problématiques dès lors que l'on commence à travailler avec des simulations de Monte Carlo. Il faut que l'on arrive à avoir une simulation de Monte Carlo rapide et efficace, donc optimisée. Idéalement, dans notre cas, la simulation de données ne devrait pas être plus longue que la prise de données équivalente avec le détecteur. Mais également, l'implémentation de la simulation doit être faite correctement pour que l'aspect stochastique de la simulation soit rigoureux et n'introduise pas de biais dans les résultats.

Dans le cadre du calcul à haute performance, une approche possible pour gagner en performance est de changer de matériel. Il existe des plates-formes de calcul plus performantes que celles utilisant des processeurs classiques (qui sont multi-cœurs), connues sous le nom d'accélérateurs matériels. On y retrouve les cartes graphiques généralistes (GP-GPU) ou encore les processeurs Intel Xeon Phi (*manycore*). Ces accélérateurs matériels permettent de gagner en performance, grâce à la parallélisation de la simulation. Mais, une fois parallélisée, il devient très critique de prêter un soin particulier à l'implémentation de la simulation stochastique, afin de rester numériquement reproductible : obtenir les mêmes résultats, quelle que soit la plate-forme d'exécution, ou encore le paradigme d'exécution (séquentiel ? Parallèle ?). Faute d'implémentation stochastique proprement gérée, il est impossible d'avoir cette reproductibilité numérique.

2 – Plan détaillé

Ce manuscrit de thèse abordera chacun des sujets définis dans le contexte, et dans la problématique à travers les différents chapitres suivants.

Dans le premier chapitre, nous reviendrons sur le contexte de la thèse, en le détaillant. Nous y expliquerons les différents processus stochastiques qui sont impliqués dans la propagation des muons à travers leur cible, mais également sur la manière dont se déroule leur génération en haute-atmosphère. Par ailleurs, nous reviendrons sur le contexte informatique, en donnant une vue

d'ensemble des différents matériels disponibles pour le calcul à haute performance à l'époque où les travaux de thèse ont été réalisés.

Dans le deuxième chapitre, nous réaliserons l'état de l'art dans les différents domaines concernés par notre problématique. Comment gérer correctement les flux stochastiques dans une simulation de Monte Carlo ? Comment optimiser une application ? Comment la profiler, même avant de l'optimiser ? Comment gérer la reproductibilité d'une application scientifique ? Nous répondrons à l'ensemble de ces questions, à l'aide des travaux de différents chercheurs sur ces problématiques. Nous les mettrons en perspectives avec nos besoins.

Ceci nous amènera donc au chapitre suivant, le troisième, dans lequel, au vu des problématiques, du contexte et de l'état de l'art, nous formulerons nos propositions pour répondre à ces problématiques dans notre contexte, voire pour proposer des méthodes plus générales qui puissent servir dans d'autres cas. C'est ainsi que dans un premier temps nous allons aborder l'optimisation en prenant le cas très précis d'une simulation de Monte Carlo qui avait été développée rapidement au début du projet ToMuVol. Nous allons d'abord présenter cette simulation, puis toutes nos propositions pour à la fois l'optimiser et obtenir une implémentation stochastique viable. Dans un deuxième temps, nous proposons une méthode, générique, qui permet de réaliser la compilation croisée d'une application, ainsi que de ses dépendances, depuis une plate-forme *multicore* vers une plate-forme *manycore*. Nous en profiterons pour recommander très fortement les plates-formes *manycore* pour des applications précises : celles dont la consommation mémoire (en terme de bande-passante) est très élevée et qui peine à fonctionner à pleine puissance sur des plates-formes classiques multi-cœurs. Nous y proposerons également l'usage d'une fonctionnalité du noyau Linux, KSM, pour réduire la consommation mémoire des applications, notamment pour les plates-formes *manycore*, où la pression de la mémoire se fait le plus ressentir. Puis, nous proposerons une méthode permettant d'implémenter un profileur, en programmation orientée aspect qui supporte à la fois le parallélisme de l'architecture matérielle sur laquelle l'application est exécutée, mais également le parallélisme de l'application elle-même. Enfin, nous conclurons les précédentes propositions de ce chapitre par un ensemble de propositions : comment implémenter une simulation de Monte Carlo qui soit reproductible, parallèle, qui passe à l'échelle et qui ait des résultats statistiquement viables. Ces propositions venant également avec des méthodes pour évaluer la reproductibilité d'une simulation. Enfin, nous proposerons un usage réfléchi des instructions vectorielles sur plates-formes *manycore* et *multicore*, résultats à l'appui. En effet, bien que toujours présentes, celles-ci peuvent plus grever les performances qu'apporter un gain.

Les deux chapitres suivants montrent la mise en œuvre des propositions rédigées dans le chapitre précédent. Dans le quatrième chapitre, nous nous pencherons d'abord sur la mise en œuvre des propositions du chapitre 3 concernant la simulation de Monte Carlo. Nous en profiterons pour exposer les bénéfices fonctionnels que nous avons pu tirer de ces optimisations (nouvelles fonctionnalités). Nous continuerons ensuite avec le détail de l'implémentation d'un profileur grâce à la programmation orientée aspect en C++. Nous expliquerons en détail son implémentation, les choix qui ont été faits, et également les résultats que nous avons pu en tirer, pour évaluer sa justesse et le bon fonctionnement de ses fonctionnalités.

Enfin, dans le cinquième et dernier chapitre, nous reviendrons sur la méta-proposition visant à implémenter une simulation de Monte Carlo parallèle, qui passe à l'échelle, dont les résultats soient statistiquement viables et qui soit reproductible. Dans ce chapitre, nous détaillerons comment nous avons mis en pratique ces propositions sur une simulation de Monte Carlo pour la propagation de muons à travers un bloc de matière. Nous reviendrons sur l'implémentation en C++ et sur les choix algorithmiques qui ont été faits pour préserver la reproductibilité, puis nous évaluerons la dite reproductibilité dans les trois cas suivants : d'une exécution à l'autre, d'un paradigme d'exécution à l'autre et en changeant d'architecture matérielle (*multicore* et *manycore*). Enfin, nous évaluerons les performances de cette simulation, son passage à l'échelle, en constatant que nous avons affaire à une simulation purement calculatoire. Nous en profiterons pour mesurer les performances du *multi-threading* sur plate-forme *multicore*.

Chapitre 1 : Contexte

1 – Introduction

Cette thèse a pris place dans le cadre du Laboratoire d'Excellence (LabEx) ClerVolc, et plus particulièrement entre deux laboratoires, membres de ce LabEx : le LPC (Laboratoire de Physique Corpusculaire de Clermont-Ferrand) et le LIMOS (Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes) à travers l'expérience ToMuVol (Tomographie Muonique des Volcans). Ces deux laboratoires se sont associés sur cette thèse en informatique en raison de son application au monde de la Physique, et plus particulièrement de la Physique des Hautes Energies.

Ce sont donc ces deux domaines que nous allons présenter dans ce chapitre, afin de donner le contexte dans lequel s'est inscrite cette thèse. Tout d'abord, nous allons donner le contexte général de l'expérience ToMuVol et ses objectifs. Puis, nous allons rentrer plus en détails sur les phénomènes physiques qui sont en jeu, ainsi que sur leur modélisation, qui sera mise en œuvre dans des logiciels de simulation de Monte Carlo. Enfin, nous nous intéresserons au contexte informatique de cette thèse, à travers un état des lieux du milieu du calcul à haute performance au moment de la réalisation de cette thèse, et à un peu d'historique pour comprendre certains choix faits aujourd'hui. Ce chapitre sera également l'occasion de poser quelques définitions utilisées par la suite, dans le reste du manuscrit.

2 – Contexte général

Comme explicité dans l'introduction, cette thèse a pris place dans le contexte de l'expérience ToMuVol. Cette expérience vise à permettre de réaliser la tomographie en trois dimensions d'édifices importants tels que des pyramides, des volcans, ou encore des cœurs de centrales nucléaires. Cette expérience est importante étant donné que ces structures en question sont des structures dont on connait finalement peu de choses : nous n'avons pas les plans des pyramides (ou alors incomplets), nous ne connaissons pas la structure interne d'un volcan (en tout cas, pas précisément), et observer comment évolue au jour le jour le cœur d'une centrale nucléaire est complexe étant donné les conditions de ces milieux.

Si ces trois éléments semblent totalement différents, ToMuVol peut permettre d'observer et de visualiser les trois et donc de répondre aux questions variées que nous nous posons au sujet des

trois. Y a-t-il des chambres secrètes dans les pyramides qui pourraient nous en apprendre plus sur la civilisation à l'origine de la construction ? Quelle est la structure interne du volcan et donc comment le magma est-il susceptible de s'y déplacer en cas d'éruption ? Quelle est la situation actuelle précise du cœur de la centrale, surtout quand on a perdu le contrôle du cœur ? Pour ce faire, ToMuVol travaille sur la mise au point d'un détecteur qui puisse être déplacé sur site pour les mesures, mais également sur les outils permettant de faire l'analyse et la reconstruction des données. De plus, étant donné que l'objectif de ToMuVol est de pouvoir observer différents types de structures, à différents moments, il faut impérativement contraindre l'expérience en terme d'encombrement ou de consommation électrique, afin que le détecteur, élément nécessaire pour pouvoir réaliser la tomographie, soit suffisamment mobile.

Plus tard dans ce manuscrit, nous ferons référence à l'objet dont nous cherchons à réaliser la tomographie comme la « cible ». Cette cible ne peut être ni trop petite, ni trop grande : comme l'expérience ToMuVol utilise les muons pour réaliser la tomographie, il est nécessaire d'avoir une cible adaptée à la particule. Typiquement, une personne sera invisible pour une tomographie muonique ; le muon est trop pénétrant et passerait à travers sans être affecté. *A contrario*, une structure trop grande, trop dense, trop épaisse ne pourrait pas être vue parce que le muon ne serait pas assez pénétrant et serait absorbé durant son parcours.

Ainsi, la tomographie muonique suit toujours le même schéma pour sa mise en œuvre : on met un détecteur qui « regarde » en direction de la cible, et qui est suffisamment loin pour pouvoir voir le flux de muons qui passe à travers la cible, mais aussi à côté. Contrairement à un scanner, il n'y a pas besoin de source de muons : ceux-ci sont générés naturellement dans la haute-atmosphère de la Terre et descendent vers celle-ci. Une fois que suffisamment de données ont été prises sous un angle, il suffit de changer le détecteur de position autour de la cible, afin de pouvoir l'observer sous un autre angle et de compléter les données. Une fois qu'il y a suffisamment de données, prises sous suffisamment d'angles, il est possible de reconstruire l'édifice.

Afin de pouvoir mettre au point son détecteur et de valider les méthodes d'analyse et de reconstruction des données, l'expérience ToMuVol s'appuie sur un volcan local français : le Puy-de-Dôme. Ce volcan du centre de la France est une cible idéale : il est endormi (et non éteint !) depuis environ 12000 ans. Il est donc stable dans le temps et dans sa forme, tout en étant local et facile d'accès. Depuis la création de ToMuVol, plusieurs campagnes de prise de données ont été réalisées, tout d'abord avec un détecteur « emprunté » puis avec le détecteur propre de l'expérience. Ces différentes campagnes de prises de données ont été réalisées à différentes périodes de l'année, mais également à différents emplacements autour du Puy-de-Dôme, afin de pouvoir mesurer le bruit

ambiant, l'impact des conditions météorologiques mais également pour avoir plusieurs prises de vues. D'autres déploiements du détecteur sont encore prévus dans le futur proche.

Parce que cette méthode est encore nouvelle (les premiers essais remontent à la fin du 20ème siècle), il est nécessaire de valider notre bonne compréhension de ce que nous observons avec le détecteur à l'aide de simulations de Monte Carlo. Autrement dit, faire converger les simulations de Monte Carlo avec la réalité observée peut aider à décrire de manière fine les phénomènes rentrant en jeu, les signaux attendus ainsi que le bruit de fond affectant les mesures. C'est ainsi que ToMuVol est une expérience grande consommatrice de simulations de Monte Carlo. Et parce que les simulations de Monte Carlo ne sont pas les plus efficaces (comparées aux simulations utilisant des modèles analytiques), elles ont un besoin de calcul à haute performance. Ce sont ces deux sujets, les modèles de propagation et le calcul à haute performance, que nous allons traiter dans les deux sections qui suivent. Dans la section 3, nous allons exposer plus en détails les processus physiques qui correspondent à la génération des muons, ainsi qu'à leur propagation à travers la matière. Nous y exposerons dans le même temps une modélisation « idéale » de la génération ainsi que de la propagation du muon. Ces modélisations sont dites idéales, parce qu'elles ne sont pas toujours complètement mises en œuvres dans les simulations qui seront exposées au cours de cette thèse, soit pour des raisons de performances ou de précision voulue dans la simulation.

3 – Le modèle de propagation des muons

3.1 - But et principes généraux des modèles de propagation

Les muons sont générés dans la haute atmosphère de la Terre. Ils sont issus de l'interaction entre les rayons cosmiques, qui sont à 86% des protons, et les noyaux des atomes constituant l'atmosphère terrestre. Les protons étant particulièrement énergétiques, il en résulte un fractionnement du noyau, et ainsi la libération d'énergie. A partir de cette énergie, des particules secondaires sont créées. En majeure partie, ces particules secondaires sont des électrons et des photons. Mais on trouve également des pions. Ces pions, instables, finiront par se désintégrer en muons. L'intégralité du processus est connue sous le nom de « gerbe atmosphérique ». En moyenne, un proton donnera quelques dizaines de muons. Néanmoins, de façon très rare, des protons extrêmement énergétiques peuvent donner jusqu'à plusieurs milliers de muons.

Individuellement, les muons sont caractérisés par leur énergie, E_μ, par leur vecteur impulsion, $\vec{p}$ (qui donne la direction du vecteur vitesse) et par leur position, donnée par un vecteur $\vec{X}$. Statistiquement, le nombre de muons produits dans l'atmosphère est caractérisé à l'aide du flux. Celui-ci est soit intégral (*i.e.*, le nombre de muons traversant l'unité d'angle solide par unité de temps), soit différentiel (*i.e.*, le nombre de muons d'une énergie et direction données traversant l'unité d'angle solide par unité de temps). A titre d'exemple, au niveau de la mer, on trouve un flux intégral de muons de 1 cm^{-2} sr^{-1} min^{-1}.

Le but principal d'un modèle de propagation des muons à travers une cible (comme le Puy-de-Dôme, par exemple) est de prédire la transmission du flux de muons atmosphériques à travers la cible. Les muons se forment dans l'atmosphère, et leur flux différentiel au niveau du sol est fourni au modèle sous la forme d'un histogramme bidimensionnel (cf. *infra*). Après avoir traversé une cible, cette distribution se trouve modifiée, à la fois en énergie et en angle d'incidence. Ces modifications sont liées aux interactions des muons avec les noyaux atomiques de la matière de la cible, et dépendent de la distribution, en termes de densité de matière, donc de composition géologique, de la cible, ainsi que de sa géométrie. Les muons de basse énergie subissent également l'influence du champ magnétique terrestre qui courbe leur trajectoire.

3.2 - La génération des muons

Les muons atmosphériques suivent une trajectoire descendante vers la surface de la Terre, arrivant sur un détecteur placé au niveau du sol de toutes les directions descendantes. Leur densité par unité d'angle solide étant gouvernée par le flux différentiel, tout se passe comme si les muons étaient générés sur la surface d'une demi-sphère centrée sur le détecteur, se dirigeant vers celui-ci en suivant les rayons de la sphère. En l'absence de cible perturbant le flux d'origine, le nombre de muons traversant le détecteur pourrait être estimé à partir d'un flux engendré sur une demi-sphère de la taille du détecteur et le contenant. Mais la présence de la cible perturbe le flux incident de muons atmosphériques et la sphère de génération doit être élargie pour l'inclure.

Les muons sont donc engendrés sur la surface de cette demi-sphère incluant la cible et centrée sur le détecteur, en accord avec le flux différentiel local des muons atmosphériques, Φ_i. Une partie des muons simulés va traverser la cible (représentée par exemple par un cône sur la figure) et c'est cette partie qui porte l'information sur la structure de la cible. Le reste des muons, qui ne traversent pas la cible lors de leur propagation, constitue ce qui est couramment appelé le flux en ciel libre et apporte une information précieuse sur le flux d'origine des muons atmosphériques, nécessaire pour mesurer la

transmittance à travers la cible, à savoir le rapport du flux mesuré sur le flux d'origine. Il est donc nécessaire de connaître le flux d'origine.

Le repère utilisé pour décrire la géométrie de notre problème est un repère direct dont l'origine est la position du détecteur (rectangle bleu sur la figure 1) et l'axe Z est la verticale locale. Les points de l'espace sont représentés, en coordonnées sphériques, par deux angles θ (le zénith) et φ (l'azimut), et par une distance à l'origine. On convient que l'angle zénithal θ est nul quand la particule vient de la verticale. L'angle azimutal φ est défini comme l'angle entre le projeté du point de départ sur le plan (O ; X ; Y) et l'axe X, les axes X et Y étant fixés arbitrairement. Les muons atmosphériques ne pouvant pas traverser notre planète, ils proviennent donc du dessus, et l'angle zénithal est compris entre 0 et $+\frac{\pi}{2}$.

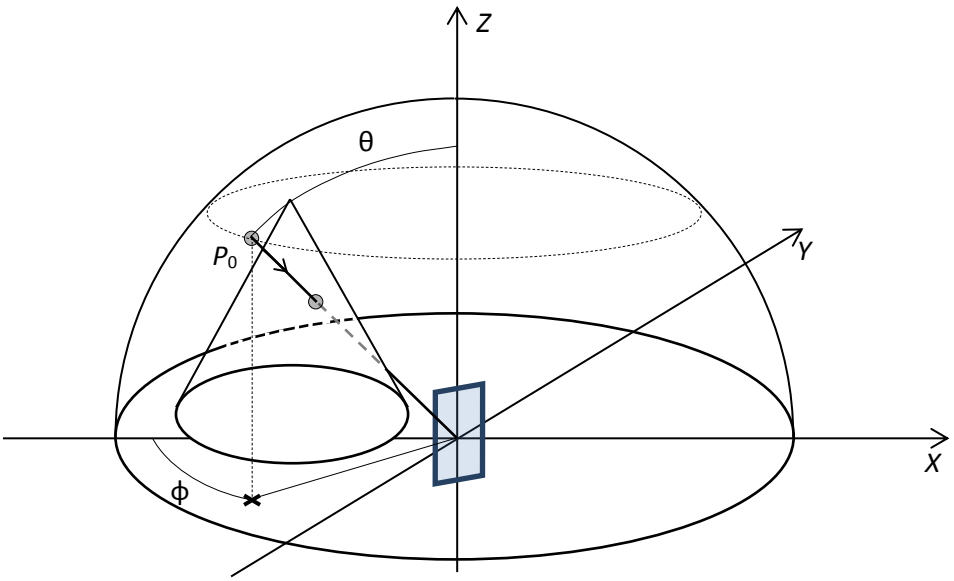

Figure 1-1 : Représentation des axes et des coordonnées utilisés pour la propagation

Si la cible a une forme simple (cône, cube, …), elle peut être modélisée de manière analytique. Sinon, elle peut être "*voxélisée*", c'est-à-dire discrétisée en *voxels* (contraction de « *volumetric pixel* »), qui sont, dans le cas le plus simple, des cubes dont le côté est égal à un pas de discrétisation constant. Dans le cas de surfaces plus complexes, l'utilisation de tétraèdres conduit à un meilleur remplissage. Un *modèle de cible* donne, en tout point de celle-ci, le contenu en matière. Comme les pertes d'énergie des muons sont régies en première approximation par la densité de la matière et comme leur dépendance envers la structure atomique intervient seulement pour de faibles corrections, le contenu en matière se réduit pour nos simulations à la description globale de la composition de la matière (atomes présents dans la matière traversée) et locale de sa densité, Nous travaillons donc sur une matière dont la composition atomique est rigoureusement uniforme mais dont la densité varie selon

les endroits dans la cible. Cette description locale de la densité se fait soit de manière analytique grâce à une fonction des coordonnées de ce point, soit grâce à un histogramme de densité en 3D basé sur la discrétisation de la cible.

Les muons interagissent avec les noyaux atomiques et, en moindre mesure, avec les électrons de la matière. Un *modèle d'interaction* entre les muons et la cible permet de simuler, en fonction de l'énergie du muon, de la densité et de la composition de matière, les pertes d'énergie ainsi que les déflections de trajectoire dues à ces interactions.

Le muon est une particule similaire à l'électron, il possède donc les mêmes propriétés et notamment, la même charge. La véritable différence avec l'électron réside dans la masse de la particule : le muon est 200 fois plus lourd. De même qu'il existe des antiélectrons (les positons), il existe des anti-muons dont la charge est positive. Les muons sont des particules instables et au repos, ils se désintègrent essentiellement en des électrons qui seront absorbés par la cible et en des neutrinos, qui passeront à travers la cible sans interagir. Dans leur référentiel propre, le temps de vie d'un muon est d'environ $2{,}2 \times 10^{-6}$ s. Dans le cas qui nous intéresse, les muons sont des particules relativistes (*i.e.,* dont la vitesse est proche de la célérité de la lumière), ainsi, leur temps de vie dans le référentiel de la Terre est plus important : il bénéficie d'un « *boost* » relativiste dépendant de son énergie, ce qui signifie qu'un muon qui possède une énergie de l'ordre de 1 TeV verra son temps de vie multiplié par un facteur 10^4. En conséquence, dans le vide, un muon d'énergie de l'ordre du GeV pourra parcourir 7 km avant de se désintégrer s'il ne lui arrive pas d'incidents impliquant une perte d'énergie. Cette distance parcourue dépend de l'énergie du muon : plus celle-ci est élevée, plus la distance que pourra parcourir le muon sera élevée. Par exemple, pour un muon dont l'énergie est de l'ordre du TeV, la distance sera de l'ordre du millier de kilomètres.

Si, lors de sa propagation dans le vide, le muon ne perd effectivement pas d'énergie, lorsqu'il se propage à travers la matière, le muon peut interagir avec le milieu qu'il traverse, et plus particulièrement avec le noyau des atomes constituant le milieu. Selon la proximité du muon au noyau et selon l'énergie du muon, trois processus différents sont possibles, tous aboutissant à une perte partielle d'énergie plus ou moins importante. Si le muon est trop éloigné du noyau de l'atome, il ne perdra que très peu d'énergie et sa trajectoire sera très légèrement défléchie, il s'agit de la diffusion coulombienne. Il pourra au passage transférer un peu d'énergie à un électron de l'atome, provoquant une ionisation de celui-ci (perte de l'électron pour l'atome). On dit que c'est une interaction élastique : le muon repart de l'atome avec l'intégralité (moins un epsilon) de son énergie initiale. Quand le muon est plus proche du noyau, il peut

perdre de l'énergie de manière radiative : soit par la production de paires électron / antiélectron, soit par le rayonnement de freinage - *bremsstrahlung* qui produit des photons (Bethe & Heitler, 1934). La perte d'énergie de façon radiative est bien plus importante qu'une perte d'énergie par ionisation. Dans ce cas d'interaction, on parle d'une interaction profondément inélastique : le muon est défléchi et repart avec moins d'énergie. Ces processus s'appliquent aléatoirement sur le muon (avec néanmoins une pondération en fonction de l'énergie de la particule - à basse énergie, l'ionisation est majoritaire tandis qu'à haute énergie, les processus radiatifs sont majoritaires (Beringer et al., 2012)). Les évènements induisant une forte perte d'énergie sont rares et dits catastrophiques. Néanmoins, plus le muon aura d'énergie, plus ils seront nombreux. De même, à trop basse énergie, le muon étant trop lent, il aura le 《 temps 》 d'interagir avec de nombreux atomes environnants et les pertes d'énergie catastrophiques seront à nouveau nombreuses.

Si l'énergie des muons leur permet, ils traversent la cible, mais peuvent être déviés, du fait des interactions avec les noyaux atomiques de celle-ci. A ces déflections liées aux interactions avec la matière de la cible, il faut enfin rajouter la déviation de trajectoire due au champ magnétique terrestre. Après propagation à travers la cible, tous les muons simulés n'atteignent donc pas le détecteur soit parce qu'ils ne pointent plus vers celui-ci, soit parce qu'ils se sont désintégrés sur leur chemin.

Inversement, les muons qui atteignent le détecteur suivant une direction (θ, φ) ne proviennent pas forcément du seul point de coordonnées (θ, φ) sur la demi-sphère, ceci ne serait le cas que si les trajectoires des muons étaient linéaires. Or les trajectoires ne sont pas linéaires à cause des interactions entre les muons et la cible, qui provoquent de multiples déflections ainsi qu'à cause de la déviation de trajectoire due au champ magnétique terrestre. Par conséquent, les muons qui atteignent le détecteur suivant une direction (θ, φ) peuvent provenir de points plus ou moins voisins.

La distribution statistique des muons en fonction de l'angle d'incidence et de leur niveau d'énergie (leur flux différentiel) n'est pas uniforme sur la demi-sphère de génération. Elle dépend essentiellement de l'énergie du muon et de son angle zénithal, mais aussi, dans une moindre mesure de l'altitude. Ainsi, pour simuler la propagation d'un muon, il convient d'abord de le générer en respectant cette distribution. Ceci peut être effectué grâce à une routine ROOT qui permet de générer un couple (E_μ, θ) conformément à une distribution bidimensionnelle (en énergie et en angle d'incidence) à l'aide, par exemple, d'un histogramme fourni en entrée au modèle. La dépendance en altitude du flux est corrigée *a posteriori* à l'aide de pondérations.

Comme explicité au début de cette section, engendrer un muon revient à engendrer sa direction, sa position et son énergie. Son énergie et l'angle zénithal étant engendrés en accord avec le flux différentiel de muons atmosphériques, il reste techniquement trois variables aléatoires à générer suivant des distributions uniformes. Pour ce faire, l'azimut φ est d'abord tiré dans son intervalle de variation [-π, π [et une fois fixé, il détermine de manière univoque la direction du muon. La position du muon dans le plan perpendiculaire à sa direction est donnée dans un repère (P_0, x, y) (*cf.*, figure 1). Dans l'idéal, il faudrait prendre, dans ce plan, une surface de génération de très grande aire. Seulement, il est inutile d'engendrer des muons qui n'ont aucune chance de se retrouver, après leur propagation en direction du détecteur, sur la surface de celui-ci. Cette aire, désignée par le terme de *surface de génération*, est donc un compromis entre les besoins d'efficacité et ceux de précision.

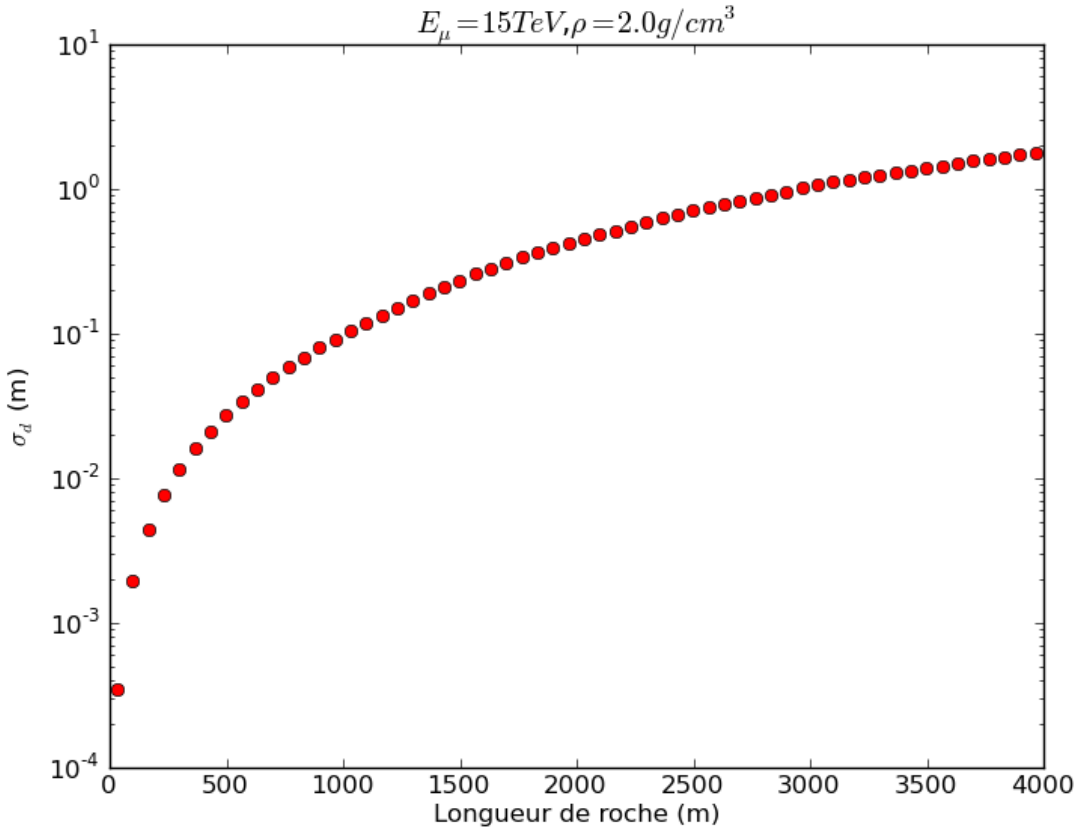

Figure 1-2 : Déviation d'un muon de 15 TeV en fonction de la distance parcourue

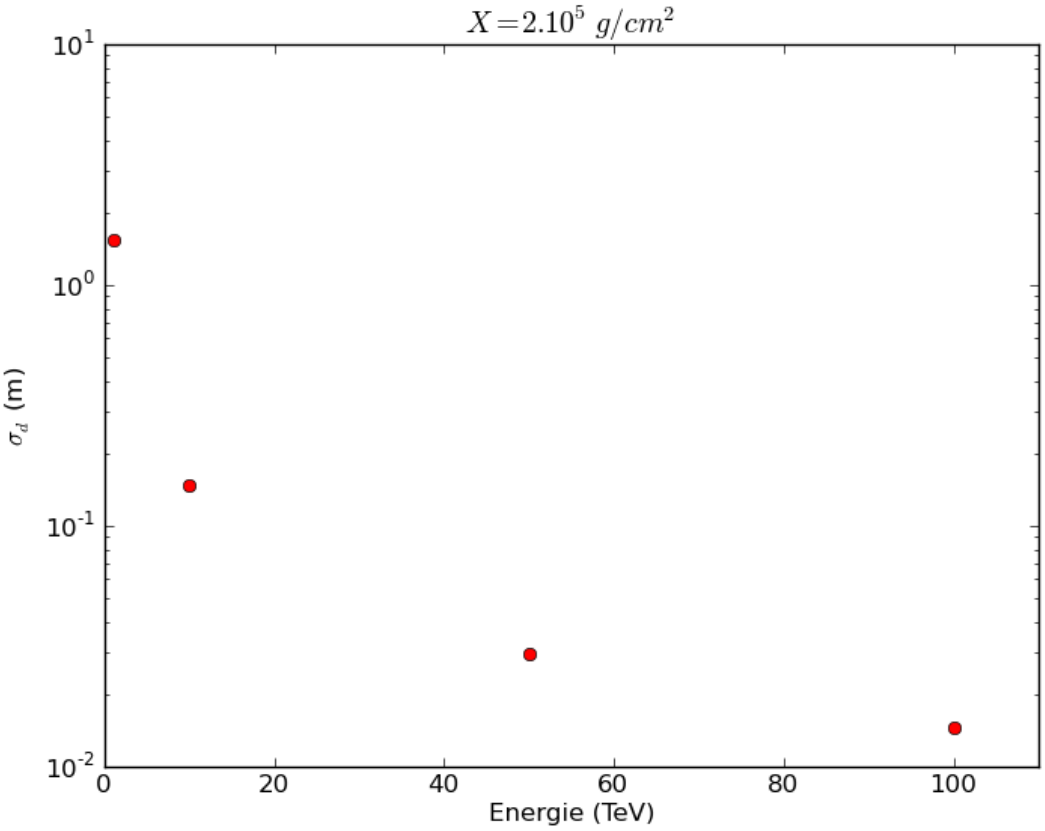

Figure 1-3 : Déviation d'un muon en fonction de son énergie initiale

En effet, l'information sur la structure de la cible est portée par les muons qui ne dévient pas beaucoup de leur direction d'origine, communément appelés muons balistiques. La reconstruction de leur direction finale, au détecteur, permet d'inférer avec une bonne approximation leur trajectoire dans la cible et mesurer ainsi, en fonction de leur taux de survie, la densité intégrée au long de leur trajectoire. Les incertitudes affectant la précision de la mesure effectuée sur cette trajectoire sont liées à la résolution du détecteur sur la direction du muon, ainsi qu'aux déflexions subites par ce muon lors de sa propagation à travers de la cible. Sur les figures 2 et 3, on peut observer la déviation des muons traversant de la roche de densité 2,00 g.cm^{-3} en fonction de la distance parcourue dans la roche (figure 2) ou de leur énergie initiale (figure 3). Sur la figure 3, on note X la quantité de matière traversée, qui correspond au produit de la distance parcourue par la densité de la matière traversée, ce qui donne, la densité étant fixée, une longueur parcourue de 1 km. On constate ainsi, que plus la longueur de roche parcourue est importante, plus un muon de 15 TeV sera dévié. De même, plus l'énergie initiale du muon est importante, moins celui-ci sera dévié sur une distance de 1 km.

Il est évident que si la surface de génération était plus petite ou de taille équivalente à la surface de la projection du détecteur sur un plan perpendiculaire à la direction du muon (c'est cette surface

perpendiculaire qui détermine « l'éclairage » en muons) la simulation serait biaisée. En effet, seuls les muons balistiques croiseraient tous le détecteur si leur énergie le leur permettait ; ceux qui auraient subi des grandes déflexions pourraient passer à côté de la surface de détection et, donc, être perdus. De plus, avec une surface de génération si petite, la simulation ne génèrerait pas des muons issus de l'extérieur de cette surface mais qui, subissant des déflections lors de leur propagation devraient quand même être détectés, avec la direction qui nous intéresse, même si leur position d'origine ne permettait pas de prévoir que leur trajectoire croiserait le détecteur. Ces derniers muons sont très importants d'un point de vue expérimental étant donné qu'ils représentent un bruit de fond. Il est donc nécessaire de les simuler afin de pouvoir estimer le bruit de fond expérimental et de pouvoir corriger le calcul de la transmittance en prenant en compte ce bruit de fond.

Pour être complètement efficace, la définition de la surface de génération devrait donc prendre en compte l'énergie du muon (c'est ce qui détermine en premier lieu l'importance des déflexions, puisqu'un muon de basse énergie est beaucoup plus sensible aussi bien au champ magnétique terrestre qu'aux interactions électromagnétiques avec les noyaux dans la matière qu'il traverse) et la topographie de la cible (plus le muon rencontre de la matière, plus il est dévié). Mais, définir une surface de génération pour chaque propagation de particule compliquerait beaucoup l'estimation du nombre de muons attendus au détecteur. En effet, les modèles de gerbes atmosphériques prédisent un flux différentiel de muons, ce qui veut dire que chaque muon engendré devrait être re-pondéré avec sa surface de génération pour retrouver le nombre attendu de muons au détecteur, afin d'éviter d'introduire un biais sur le flux mesuré dans la simulation. Le premier choix, celui de la facilité, est donc d'avoir une surface de génération indépendante de l'énergie du muon et ayant seulement une dépendance binaire à la topographie : si l'angle zénithal permet de prévoir que le muon ne traversera pas la cible (flux ciel libre), la surface de génération est plus petite que dans le cas contraire. A noter enfin, qu'une fois la fenêtre calculée, celle-ci doit subir une translation pour prendre en compte la charge électrique du muon, et donc sa déviation magnétique (la translation est donc effectuée dans le sens inverse pour les anti-muons). En effet, contrairement aux diffusions coulombiennes qui élargissent la distribution des positions d'un faisceau de muons sans décaler sa valeur moyenne, le champ magnétique agit non seulement sur la largeur de la distribution, mais également sur sa moyenne, puisqu'il affecte de manière cohérente tous les muons avec la même charge et la même direction.

4 – Le contexte du calcul à haute performance

En raison de l'usage intensif de simulations de Monte Carlo, la question du calcul à haute performance s'est immédiatement posée, notamment en raison des performances plutôt mauvaises

de la simulation initialement développée pour ToMuVol (détails au chapitre 3). Il y a plusieurs approches pour le calcul à haute performance. Outre la nécessité claire d'optimiser les applications dans tous les cas, on peut envisager trois méthodes différentes pour tenter de gagner en performances : utiliser un cluster de calcul, c'est-à-dire un agrégat local de nœuds de calcul, ou utiliser la grille de calcul qui est (de façon simplifiée) un agrégat mondial de nœuds de calcul, exploités sur internet, ou encore, dernière possibilité, changer d'architecture cible pour l'exécution et utiliser des accélérateurs matériels.

Dans le cadre de ToMuVol, étant donné l'architecture de la première simulation, qui s'appuyait sur un logiciel en Java pour sa partie calculatoire, la piste de la grille de calcul a été rapidement abandonnée. Le déploiement d'un tel logiciel, avec l'intégralité de ses dépendances était très compliqué. Une piste utilisant CDE (Guo & Engler, 2011) a été évoquée, mais elle déplaçait la difficulté plus qu'elle ne réglait le problème sous-jacent.

Nous nous sommes donc intéressés aux deux autres solutions : le cluster et les accélérateurs matériels. Dans le cadre de ToMuVol, les clusters sont classiques et leur nombre de nœuds limités. Rien de comparable par rapport aux clusters représentés dans le TOP500[3]. Ce classement vise à déterminer qui possède le cluster qui est capable d'avoir le rendement en TFLOP/S le plus élevé. Dans notre cas, l'architecture de ces nœuds était purement x86, à base de processeurs Intel Xeon. C'est d'ailleurs l'architecture de calcul la plus répandue à l'heure actuelle ; le TOP500 en est la preuve, si l'on prend les 20 premiers clusters, 15 sont à base de processeurs x86_64.

Néanmoins, on peut constater dans le TOP500 l'émergence forte des accélérateurs matériels. Au dernier classement (juin 2015), le premier, Tianhe-2 utilise un modèle de l'accélérateur matériel d'Intel le Xeon Phi, et le second, Titan, utilise un modèle d'accélérateur matériel de chez NVIDIA, les K20x.

Le premier à être arrivé sur le segment des accélérateurs matériel est NVIDIA. Normalement fondeur de processeurs graphiques (GPU, *Graphical Processor Unit*), NVIDIA s'est aperçu de leur potentiel dans le monde du calcul à haute performance : un GPU est une forme d'implémentation idéale du SIMD (*Single Instruction, Multiple Data*). Cette catégorie d'architecture signifie qu'une unique instruction pourra être exécutée en même temps sur plusieurs données : et réaliser ainsi un calcul vectoriel. Elle s'oppose au SISD (*Single Instruction, Single Data*) qui correspond aux processeurs classiques : une instruction unique, sur une unique donnée. Si les processeurs contiennent quelques instructions SIMD, leur performance n'a rien à envier avec la capacité de calcul d'une carte

[3] http://www.top500.org/

graphique, qui d'ailleurs contrairement à un processeur classique, calcule en virgule flottante par défaut. C'est ainsi que NVIDIA est arrivé à la création des GP-GPUs (*General Purpose GPU*), qui sont des GPU dont le seul et unique but est le calcul (scientifique) ; ces cartes graphiques ne possèdent même plus de sortie vidéo. Depuis sa première mise en œuvre à travers les cartes accélératrices Tesla fin 2008, la technologie a beaucoup évolué chez NVIDIA qui en est aujourd'hui à sa troisième génération, Kepler. En plus de ses cartes accélératrices, NVIDIA a proposé une extension au langages de programmation C et C++ ainsi qu'un nouveau paradigme de programmation appelé CUDA (*Compute Unified Device Architecture*) (Garland et al., 2008). Cette architecture permet de développer des applications parallèles pour GP-GPU en s'appuyant sur les langages C et C++ qui sont étendus notamment avec la notion de « *kernel* ». Le *kernel* est l'unité parallèle qui sera exécuté sur le GP-GPU, tandis que le reste du code restera local aux processeurs du nœud de calcul.

Cette technologie GP-GPU a permis à NVIDIA de s'imposer face à Intel dans le marché du calcul à haute performance. Notamment par des publications de l'époque 2008-2009, quelques peu naïves (ou plutôt savamment orchestrées de façon commerciale) où des scientifiques comparaient les performances d'un unique cœur de processeur face à un GP-GPU entier. Forcée de réagir, la société Intel a réagi de deux façons. Dans un premier temps, les ingénieurs d'Intel ont publié un article appelant aux « *fair comparisons* » (Lee et al., 2010). C'est d'ailleurs dans ce cadre que se placeront nos propres évaluations de performance dans ce manuscrit : nous exploiterons les deux architectures au maximum avant de comparer leurs performances respectives, afin de ne pas biaiser les observations. En parallèle, la deuxième réaction d'Intel a été de mettre au point son propre accélérateur matériel, le Xeon Phi. S'il se présente comme un GP-GPU dans une carte d'extension PCI, la comparaison s'arrête là. Le Xeon Phi s'appuie sur une architecture matérielle nommée Knight Corner (ou k1om) (Chrysos, 2012). L'architecture du Xeon Phi reprend une architecture de type x86_64 avec quelques différences tout de même. Elle possède moins d'instructions qu'une architecture x86_64 et possède ses propres instructions vectorielles. De plus, l'architecture k1om n'est pas câblée comme un CPU standard. De fait, un Xeon Phi possède bien plus de cœurs physiques qu'un CPU classique (60 ou 61) et chaque cœur physique possède quatre threads matériels « *hardware threads* » tandis qu'un cœur physique de CPU se contente de l'hyper-threading (Marr, 2002) qui lui permet d'avoir deux cœurs logiques. De même, le pipeline d'instructions d'un cœur Xeon Phi est fortement réduit à cinq étapes seulement contre une vingtaine pour un CPU actuel. Par ailleurs, les caches plus importants sur un Xeon Phi sont organisés en forme d'anneau, ce qui fournit une interconnexion entre les différents cœurs. En contrepartie, alors que les CPUs actuels ont une fréquence d'horloge entre 2 GHz et 4 GHz, un Xeon Phi dépasse à peine les 1 GHz (1,2 GHz pour les modèles actuels les plus performants). Enfin, contrairement aux

GP-GPUs, les Xeon Phi peuvent embarquer une quantité importante de mémoire RAM, entre 8 Go et 16 GB selon les modèles. Ce dernier propos est néanmoins à nuancer avec les dernières cartes GP-GPU proposées par NVIDIA, qui étant bi-GPU, embarquent finalement plus de RAM qu'un Xeon Phi. En effet, si l'on considère un GP-GPU Kepker K80, il embarque au total 24 Go de RAM (soit plus qu'un Xeon Phi), cependant, il est bi-GPU, donc, chaque GPU ne dispose que de 12 Go, ce qui est moins qu'un Xeon Phi à 16 Go. Il apparaît donc clairement que comme les GP-GPUs, le Xeon Phi n'est pas prévu pour être exploité pour une application séquentielle. Ainsi, comparer les performances d'un cœur de Xeon Phi et de celle d'un cœur de processeur classique ne présente guère d'intérêt.

Le Xeon Phi a été pensé pour éviter certains défauts que présentent les GP-GPUs. Notamment, le Xeon Phi peut être utilisé de deux manières différentes. En effet, le Xeon Phi est en réalité un nœud de calcul dans une carte d'extension PCI. Ceci signifie que le Xeon Phi fait tourner son propre système d'exploitation Linux et peut par exemple être directement utilisé *via* SSH (*Secure SHell*) comme une machine SMP (*Symmetric MultiProcessing*). Ce mode est appelé mode natif (ou « native mode »). Autrement, le Xeon Phi peut être utilisé comme un GP-GPU avec une application qui tourne sur l'hôte et qui exécute certaines portions parallèles sur le Xeon Phi. Dans ce cas, le Xeon Phi est utilisé en « *offload mode* ». Le mode natif simplifie énormément l'utilisation du Xeon Phi, il suffit juste de l'exploiter comme un nœud de calcul standard, avec son architecture parallèle propre. Cela signifie notamment que le Xeon Phi supporte nativement OpenMP ainsi que MPI, chose qui était impossible avec les GP-GPUs. Par ailleurs, pour éviter d'imposer un nouveau langage de programmation comme CUDA, Intel a fait le choix d'implémenter le support de l'*offload* mode uniquement *via* des `pragmas` de compilation. L'objectif annoncé est de simplifier le portage d'une application pour CPU vers le Xeon Phi. Néanmoins, à l'heure actuelle, seuls les compilateurs Intel supportent le Xeon Phi. Ce qui signifie que seuls les langages de programmation C, C++ et Fortran peuvent être utilisés pour développer pour le Xeon Phi, et ce quel que soit le mode utilisé.

Nous reviendrons sur les outils permettant de travailler avec les Xeon Phi, que ce soit pour le développement ou pour le profilage dans le chapitre suivant présentant l'état de l'art. Nous présenterons également quelques-uns des outils qui permettent de travailler sur les deux types d'accélérateurs matériels présentés. Cet état de l'art sera aussi l'occasion de présenter des études des performances comparées des deux architectures. A ce titre, dans cet état de l'art et dans l'intégralité du manuscrit, nous utiliserons deux termes très importants du monde du calcul à haute performance dont nous tenons à fixer précisément la définition. Le gain de performance (ou en anglais, « *speedup* ») est la mesure de l'amélioration des performances d'une application. Etant

donnée la performance initiale d'une application (en terme de temps de calcul), T_i, et sa performance une fois améliorée, T_a, le gain de performance est calculé ainsi : $\frac{T_i}{T_a}$. Si le résultat est supérieur à 1, on a véritablement un gain de performance : l'application améliorée est plus performante, autrement, on a une perte de performance. On notera généralement les gains de performances avec la notation « X » : par exemple, un gain de performance de 2X signifie que la version améliorée est deux fois plus performante que la version initiale. A l'aide de cette définition du gain de performance, nous pouvons définir l'efficacité d'une application. Etant donné le gain de performance que l'on mesure en utilisant p cœurs du processeur S_p, et le gain de performance théorique maximal que l'on pourrait avoir sur ces p cœurs S_m, alors l'efficacité est : $\frac{S_p}{S_m}$. Plus le résultat est proche de 1, plus l'application est efficace. Si elle est égale à 1, l'application est dite « linéaire ». S'il est supérieur à 1, l'application est dite « super linéaire ».

Dans le cadre de notre thèse, nous avons très vite évacué la possibilité de travailler sur GP-GPU : nos codes de simulation, très complexes, s'appuient sur des cadriciels encore plus complexes avec au total quelques millions de lignes de code, pour lesquels des portages existent pour les GP-GPUs, mais de manière très limitée et partielle en terme de fonctionnalités. L'objectif de cette thèse n'était pas de porter ces cadriciels vers GP-GPU (la tâche aurait été particulièrement ardue pour une seule personne). Nous nous sommes plutôt tournés vers le Xeon Phi, grâce à sa facilité d'utilisation : selon Intel, il suffit de compiler l'application pour le Xeon Phi, et « ça marche ». C'est notamment ce que nous explorerons dans le reste de ce manuscrit.

5 – Conclusions

Dans ce chapitre, nous avons posé le contexte de cette thèse. Dans un premier temps, nous avons expliqué ce qu'était la muographie, sa mise en œuvre au sein de l'expérience ToMuVol et l'objectif de cette dernière de vouloir réaliser la tomographie de volcans, ou d'autres grands édifices. C'est ainsi que nous avons mis en évidence le besoin d'avoir des simulations de Monte Carlo efficaces et donc, le besoin en calcul à hautes performances. Nous avons alors présenté les différentes stratégies mises en œuvre dans le monde du calcul à haute performance. Nous avons rapidement étudié leur adéquation avec les besoins de l'expérience. C'est ainsi que la grille de calcul a été écartée. En revanche, nous avons centré notre étude sur les accélérateurs matériels : nous nous sommes

d'abord intéressés à ceux de NVIDIA, puis nous avons présenté le nouvel accélérateur matériel créé par Intel, le Xeon Phi. Nous en avons justifié notre utilisation, *a priori*, pour les travaux de thèse, en écartant les GP-GPUs. Enfin, ce chapitre a été l'opportunité de poser quelques définitions relatives au calcul à haute performance, concernant des termes qui seront utilisés dans l'ensemble du manuscrit. Nous avons également expliqué la rigueur scientifique avec laquelle nous avons effectué nos évaluations de performances : nous n'avons fait que des comparaisons honnêtes, en exploitant au maximum les architectures avant de les comparer.

Dans le chapitre suivant, concernant l'état de l'art, nous présenterons les outils de profilage et d'optimisation pour les deux architectures cibles sur lesquelles nous avons travaillé tout au long de ce travail de thèse : le processeur x86 et l'accélérateur matériel Xeon Phi.

Chapitre 2 : Etat de l'art

1 – Introduction

Une application, que ce soit dans le calcul hautes performances ou non, peut se révéler particulièrement lente alors que ses performances sont critiques. On peut avoir besoin de ses résultats en temps réel, ou en un temps nettement plus court que la production équivalente d'un système naturel (dans le cas d'une simulation complexe de météorologie par exemple). Il peut aussi être nécessaire que les résultats signifient encore quelque chose au moment où ils deviennent disponibles (obsolescence, simulation en finance ou simulation de phénomènes naturels).

Une application peut être gênée dans son exécution par plusieurs types de problèmes. Si le problème est principalement lié à un calcul intensif, alors il faut que l'application puisse efficacement utiliser le processeur et ses instructions. On parle d'ailleurs alors d'applications « *CPU-bound* » (bornée par l'aspect processeur, donc liée à l'aspect calcul). On s'attachera alors à utiliser les instructions les plus efficaces et les moins coûteuses en cycles (l'unité de mesure du processeur). Si le problème fait un usage intensif des accès mémoires, alors l'application a un rapport accès mémoire / calcul élevé. Dans ce cas précis, on parle alors d'application « *memory-bound* » (limitée par les performances de la mémoire) (McCalpin, 1999). On travaillera alors à optimiser les accès mémoire de l'application pour qu'ils coûtent le moins cher. En effet, un processeur possède plusieurs types de mémoires (registres, caches, RAM). Les coûts des accès sont également comptabilisés en cycles processeur. Ainsi, sur un processeur récent Intel i7 (Levinthal, 2009) (Büttner & Weidendorfer, 2013), un accès à la mémoire la plus rapide coûte 1 cycle, un accès au cache de niveau 1 (L1) coûte 4 cycles, un accès au cache de niveau 2 (L2) coûte 10 cycles. Dès lors que l'on quitte ces zones mémoires dites locales au processeur, le coût explose. Le coût pour un accès au cache de niveau 3 (L3) est entre 40 cycles et 75 cycles selon que l'on soit dans le meilleur ou dans le pire des cas. Il est entre 200 et 300 cycles pour des données en RAM et supérieur à 10^7 cycles pour un accès au disque dur. Dans ce dernier cas, il se trouve qu'un accès à l'information implique des mécanismes d'entrées sortie (ici le disque dur). On peut donc aussi distinguer les applications limitées par les entrées sorties en les classant « *I/O-bound* ». La problématique de localité de la mémoire et, dans un cadre plus général, celle de l'accès à l'information, prend alors tout son sens. Plus la donnée sera proche du processeur (voire proche des unités de calculs dans le processeur) plus rapide sera l'accès à celle-ci.

Dans ce chapitre, on s'intéressera donc à la problématique de l'optimisation des applications pour en obtenir les meilleures performances possibles. Ainsi, dans un premier temps, nous présenterons

de nombreux outils permettant d'analyser les performances de l'application (profileur) et d'en extraire les informations sur les zones identifiées comme présentant de faibles performances. Dans une seconde partie, nous présenterons différentes méthodes permettant d'optimiser une application, que ce soit pour sa capacité de calcul brut, ou bien pour sa capacité à accéder aux données. Enfin, dans une troisième et dernière partie, nous nous intéresserons aux conséquences de l'optimisation, tant en terme de performances maximales atteignables qu'en terme de reproductibilité des résultats de l'application.

2 - Profilage d'applications

Afin de pouvoir optimiser l'application, il convient donc de savoir si elle est *memory-bound* ou *CPU-bound*, ou de façon générale, de savoir si elle est efficace et exploite correctement le serveur de calcul. Il est donc nécessaire de la profiler, afin d'avoir une cartographie précise de l'application. Un but est d'avoir des informations sur l'ordre d'exécution dans l'application (quelles fonctions appellent lesquelles ? quand ? combien de fois ?), ce qui permet de définir un arbre des appels, et de retracer tout l'historique du fonctionnement de l'application, mais, l'autre but est également de pouvoir associer un coût à chacun de ces appels de fonction, voire, quand on est suffisamment précis, un coût à chacune des instructions constituant la fonction. Ceci permet, à la fin, d'avoir dans l'arbre des appels, les zones de l'application qui ont le coût le plus élevé et/ou celle qui sont le plus souvent appelées. Il est dès lors possible d'optimiser l'application. Attention néanmoins : optimiser les fonctions ayant le coût le plus élevé n'est pas nécessairement le plus judicieux. Si cette fonction n'est appelée qu'une fois dans toute l'exécution de l'application, le gain ne sera pas aussi important qu'une fonction avec un coût un peu moins élevé mais appelée un grand nombre de fois.

Jusqu'ici, nous avons parlé de coût et non de durée d'exécution ou de temps processeur pour la mesure effectuée pour chaque fonction ou instruction. En effet, le but du profilage est bien de mesurer un coût. Et selon l'optimisation que l'on cherche à réaliser, le coût n'est pas le même. On peut mesurer l'occupation mémoire, le temps processeur, le nombre d'instructions processeur, le nombre de « *cache-miss* », etc. Ce coût est mesuré grâce à différents compteurs disponibles fournis soit par le système d'exploitation soit par le matériel lui-même. Il existe plusieurs méthodes pour profiler une application, et toutes ne s'appuient pas sur les mêmes compteurs et ne proposent donc pas les mêmes coûts dans leur verdict. Il est donc de bon usage d'avoir un premier profilage suffisamment générique pour avoir assez de données et pouvoir établir si l'application est *CPU*

bound ou *memory bound*. En fonction des cas, on pourra sélectionner les compteurs qui présentent le plus d'intérêt et le plus d'informations sur la problématique évaluée.

Avant toute tentative de profilage, il est nécessaire de travailler sur une application déjà optimisée par le compilateur et qui fonctionne dans les conditions de production, c'est-à-dire sans code de débogage et avec les options d'optimisation du compilateur. Il est néanmoins recommandé de laisser les symboles de débogage dans l'application et ses dépendances, ou en tout cas, de les avoir dans des fichiers séparés, afin de pouvoir lire les données issues du profilage et pouvoir les relier à leur emplacement dans le code et donc se faciliter le travail d'analyse des données.

2.1 – Profilage grâce au compilateur

Une première technique d'optimisation liée au profilage est de s'appuyer sur le compilateur directement. En effet, un des coûts majeurs lors du fonctionnement de l'application est dans les sauts effectués lors de l'exécution du code. Le code d'une application est linéaire, or, pour gérer les structures logiques telles que des conditionnelles ou des boucles il est nécessaire de sauter à d'autres positions dans le code. Cette opération est coûteuse en cycles (vidage du pipeline), et doit être évitée au maximum. Par défaut, le compilateur a peu de chance de connaître quelles seront les branches empruntées dans la majorité des cas et qui provoqueront donc des sauts dans le code. Il est possible d'aider le compilateur directement dans le code, notamment avec le compilateur GCC et son instruction `__builtin_expect`. Par exemple, dans le cas d'un appel à `open`, on récupère le descripteur de fichier dans une variable `fd`, il est peu probable que l'ouverture échoue en temps normal, on utilisera donc cette instruction comme dans l'extrait de Code 1.

```
int fd = open(pathname, flags);
if (__builtin_expect((fd < 0), 0)) /* if (fd < 0) */
{
    /* gestion erreur */
}
/* Exécution normale */
```

Code 2-1 : Utilisation de l'instruction `__builtin_expect` de GCC

Ceci indiquera au compilateur que le bloc du `if` sera accessible *via* un saut tandis que le reste du code, non. Ainsi, il n'y aura une pénalité liée à ce saut que dans des cas rares. Maintenant, écrire ces

indications à chaque fois rajoute une lourdeur de développement supplémentaire qui rendra l'ensemble du code moins lisible, d'autant plus qu'il y a de nombreux cas où la prédiction de branche n'est pas immédiate. Il est donc possible de s'appuyer sur le compilateur pour à la fois évaluer quelles sont les sauts à supprimer et pour générer proprement le code avec ces sauts en moins. On utilise alors la technique de « l'optimisation dirigée par les profils » (Pettis & Hansen, 1990), en anglais « *profil guided optimisation* » (PGO). On réalisera d'abord une compilation en indiquant au compilateur de rajouter du code de profilage dans l'application compilée, pour évaluer les embranchements du code. Ensuite, on exécute le code dans les conditions normales de fonctionnement. Cette exécution va permettre au code de se profiler et va générer des données. On pourra alors réaliser une nouvelle compilation du programme en demandant au compilateur d'utiliser les données pour optimiser les sauts et les positions du code dans le binaire final. Les grands compilateurs du marché actuels supportent ce mécanisme d'optimisation, qui n'a néanmoins rien de standard. Chaque compilateur l'implémente et le gère à sa façon. Ainsi, GCC le supporte grâce aux options « `-fprofile-generate` » et « `-fprofile-use` », ICC grâce aux options « `-prof_gen` » et « `-prof_use` », MSVC grâce aux options « `/GL /LTCG:PGINSTRUMENT` » et « `/LTCG :PGOPTIMIZE` », Clang grâce aux options « `-fprofile-instr-generate` » et « `-fprofile-instr-use` ». Un gros avantage de cette méthode est quelle est supportée sur toutes les plates-formes matérielles sur lesquelles le compilateur est disponible. Ce qui n'est pas forcément le cas pour les autres techniques de profilage.

Pour le profilage qui nécessitera une analyse manuelle et des changements dans le code source ensuite, on peut s'appuyer en tout premier lieu sur le compilateur pour le réaliser. En effet, on peut demander à GCC de modifier le code pour y ajouter des instructions de profilage du code. Ceci se fait à la compilation, grâce à l'option « `-pg` ». Lors de la prochaine exécution de l'application, elle sera profilée et un fichier binaire contenant les résultats du profilage sera produit. Ce profilage s'intéresse au temps passé dans les fonctions. En effet, ce profilage simple consiste simplement à interrompre l'application à intervalle réguliers et à regarder dans quelle fonction elle est au moment de chaque interruption. On peut ainsi, statistiquement, établir le temps passé dans chaque fonction. C'est également ce que l'on appelle un profilage asynchrone tel que défini dans (Malony, Mellor-Crummey, & Shende, 2011). Il est asynchrone parce qu'il ne suit pas le flux d'exécution standard du programme mais vient régulièrement l'interrompre. Enfin, pour extraire les données du fichier binaire produit par le profilage, il faut utiliser l'outil `gprof` (GNU Profiler) (Graham, Kessler, & Mckusick, 1982) qui produira une sortie texte avec le temps passé dans chaque fonction. Cette méthode de profilage, si elle est simple à mettre en œuvre et à utiliser pour du code *mono-threadé* ne supporte pas les applications *multi-threadées* et produirait dans ce cas un fichier binaire avec des

données incohérentes. Par ailleurs, les relevés de coûts sont simplistes et peu précis. Il est néanmoins possible d'avoir le profilage fonction par fonction ou ligne par ligne (Fenlason & Stallman, 1988).

Pour les autres méthodes de profilage que nous présenterons ensuite, nous n'utiliserons plus de fonctionnalités spécifiques au compilateur, mais des bibliothèques, des applications ou des portions de code tierces pour le profilage. Nous distinguerons néanmoins deux cas de figures : le profilage où il est tout de même nécessaire de recompiler l'application pour pouvoir la profiler et les cas où le profilage vient se greffer sur l'application déjà compilée.

2.2 - Profilage d'une application sans recompilation

Un outil open source ne requérant aucune recompilation de l'application est `callgrind` (Weidendorfer, Kowarschik, & Trinitis, 2004). Celui-ci fait partie de l'ensemble des outils proposés par le logiciel `valgrind` (Nethercote & Seward, 2003). `Valgrind`[4] exécute l'application dans un environnement virtuel qu'il contrôle et donc, sur lequel il peut effectuer des mesures, selon l'outil qu'on utilise. Par défaut, `valgrind` fera un contrôle de la mémoire (allocations, libérations, dépassements de zones, etc.). Néanmoins, d'autres outils existent. Certains permettent de vérifier l'exécution correcte d'une application parallèle (ressources critiques, exclusions mutuelles, etc.) comme `helgrind` (Valgrind-project, 2007) ou encore l'utilisation efficace des caches de la machine, comme `cachegrind`. De son côté, `callgrind` va profiler l'application et établir un graphique des appels (hiérarchie des appels dans l'application). Les coûts profilés peuvent être en temps système, mais par défaut, `callgrind` travaille avec le nombre d'instructions. C'est-à-dire que pour chaque fonction et pour chaque ligne de code, il donnera le nombre d'instructions effectuées. A partir de cela, il peut également fournir une indication (et non une mesure !) de la quantité de cycles consommés. Parce que `valgrind` exploite un environnement virtuel, les architectures qu'il supporte sont limitées. Par exemple, `valgrind` ne peut pas profiler une application compilée pour Xeon Phi. De même, il ne supporte pas les nouveaux jeux d'instructions des processeurs dès leur disponibilité sur le marché. La sortie de `valgrind` est un fichier texte au formalisme très précis. La précision des données collectées sera augmentée par l'utilisation des symboles de débogage lors de la compilation et par la présence du code source lors de la lecture des données issues du profilage. A l'issue du profilage, pour pouvoir exploiter correctement les données récoltées par `valgrind`, il est nécessaire d'utiliser un outil tiers qui puisse correctement les

[4] http://www.valgrind.org/

corréler et les mettre en forme. Un outil disponible pour ce faire est `Kcachegrind`, de la suite logiciel KDE. Il est également à l'heure actuelle, le plus complet et donc, le plus largement utilisé.

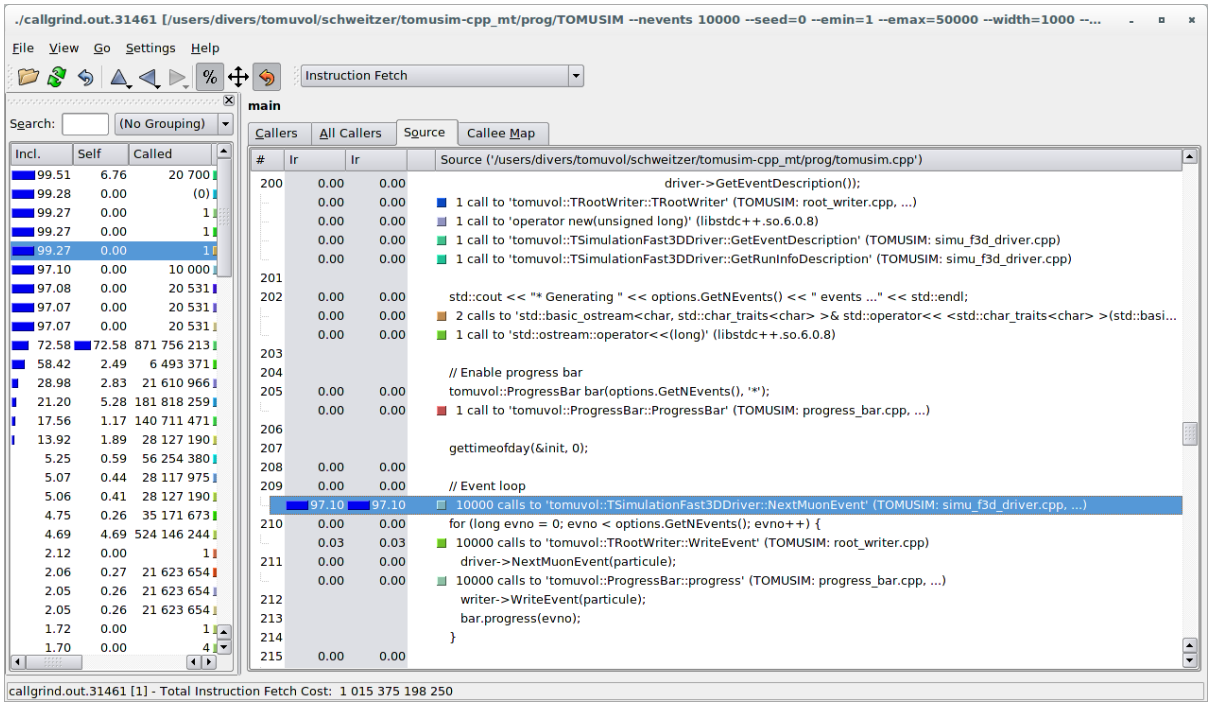

Figure 2-1 : une vue de `Kcachegrind`

La figure 1 présente l'une des vues de `Kcachegrind`. Sur celle-ci, on voit notamment le code de l'application qui a été profilée et le coût associé à chaque ligne de code (en instructions). Il est également précisé le nombre d'appels qui a été réalisé afin, toujours, de concentrer les efforts sur les fonctions qui prennent du temps et sont souvent appelées. A gauche, on voit également la liste des différentes fonctions appelées (quelles soient du programme ou d'une bibliothèque externe comme la `libc`) et leur coût associé pour un appel ainsi que pour le cumul des appels. Ceci fournit une indication très rapide des zones du code où il faudra produire des efforts. On peut voir également les onglets présents qui permettent de naviguer entre les autres vues de `Kcachegrind` qui fournissent notamment de façon graphique l'arborescence complète du programme exécuté, toujours avec les coûts inclusifs associés.

Une autre approche du profilage est utilisée par `VTune` (J. H. Wolf, 1999), le profiler commercial de Intel. Pour profiler une application avec `VTune`, là non plus, il n'est pas nécessaire de la recompiler. Mais, contrairement à `valgrind`, `VTune` ne lance pas d'environnement virtuel, mais va lire les compteurs de performances fournis par les processeurs (Bhandarkar & Ding, 1997). Ceux-ci fournissent en effet des relevés très précis sur l'utilisation du processeur. On peut y trouver des informations sur la bonne utilisation de tous types de caches, notamment sur la bonne utilisation

des caches TLB (*Translation Lookaside Buffer*) (Stravers & Van De, 2004), ou encore sur la bande-passante consommée par l'application. `VTune` réalise tout le travail de profilage et de mise en forme des données collectées, donc il n'y a pas besoin ici d'un logiciel tiers comme pour `valgrind`. De plus, `VTune` étant un produit Intel, il supporte parfaitement les plates-formes Intel, y compris les architectures du type Xeon Phi, dès leur disponibilité sur le marché.

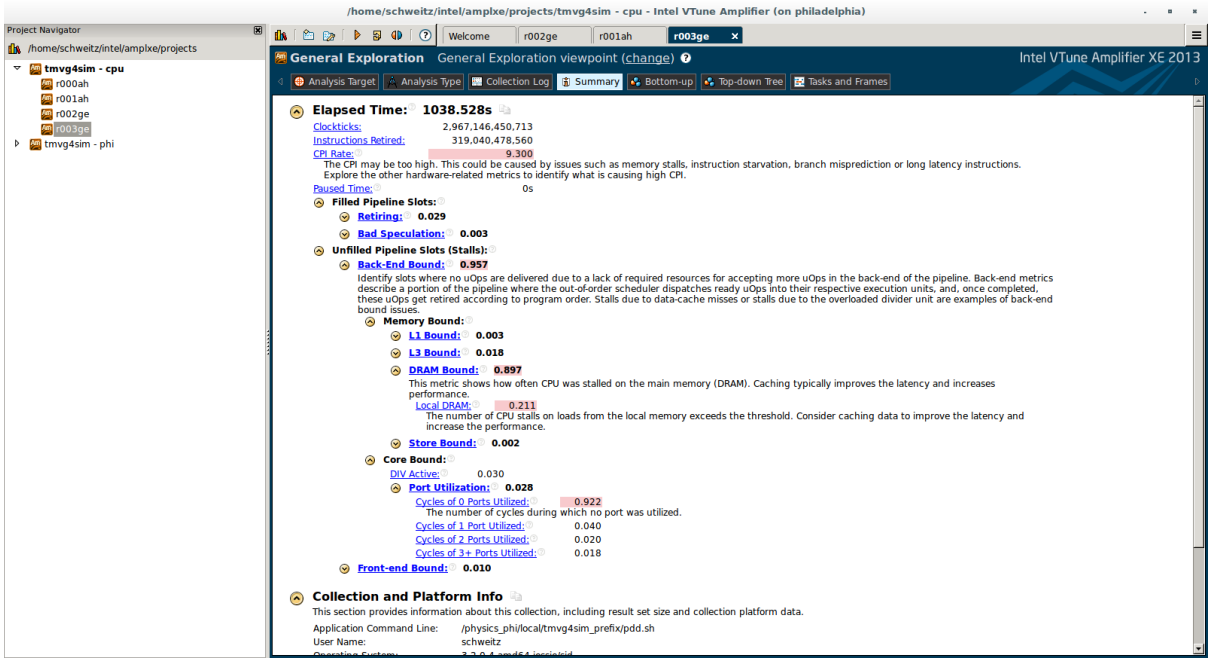

Figure 2-2 : Une vue de `VTune`

La Figure 2 présente une vue de `VTune`. On y voit les résultats du profilage d'une application *via* la méthode de « *General Exploration* ». Grâce à cette méthode, `VTune` tentera de récupérer un maximum d'informations sur l'état de la machine lors de son exécution. Pour aider à identifier les problèmes, `VTune` met en rouge les compteurs qui ne sont pas bons. Ici, `VTune` fonctionne également à l'aide d'onglets qui permettent de changer de vue et par exemple, d'avoir également l'affichage du comportement de l'application dans le temps, par thread ou encore ligne par ligne. Si les données fournies par `VTune` sont bien plus précises que celles données par `valgrind`, il est également nécessaire d'avoir des connaissances bien plus pointues sur l'architecture utilisée pour pouvoir en avoir une lecture correcte et qui ait un sens. La documentation de `VTune` fournit également de plus amples détails pour aider à la compréhension de son fonctionnement.

Une troisième approche de profilage est utilisée par `GPerfTools` (Google Performance Tools)[5]. Ces outils de mesure de performances tiennent dans une ou plusieurs bibliothèque(s) partagée(s) (« *shared objects* »). Il y a deux utilisations possibles. Ou bien l'on recompile l'application en la « liant » à cette bibliothèque partagée, ou bien on force le chargement de la bibliothèque au démarrage de l'application. On ne s'intéressera pas au premier cas, trivial, qui ne rentre pas dans le spectre couvert par ce paragraphe, puisqu'il s'agit de recompiler l'application. Le deuxième requiert de forcer l'application à charger la bibliothèque au démarrage, étant donné qu'elle n'a pas été compilée avec. Linux fournit un mécanisme pour y parvenir : la variable d'environnement « `LD_PRELOAD` ». *Via* celle-ci, il est possible de demander à Linux d'attacher des bibliothèques partagées à une application lors de son chargement. Les éléments statiques de la bibliothèque seront construits lors de son chargement et ses fonctions exportées seront également disponibles pour l'application. Ceci permet à `GPerfTools` de fournir deux profilages différents. Une des bibliothèques fournies exporte les fonctions d'allocation mémoire standard de Linux (`malloc`, etc.) ce qui permet de profiler l'utilisation de la mémoire par l'application, comme le ferait `valgrind` avec un outil de vérification mémoire. La deuxième bibliothèque proposée par Google, permet de profiler l'utilisation du CPU. Ici, le profilage est réalisé de façon asynchrone, de la même façon que l'option « `-pg` » du compilateur GCC. L'application est interrompue fréquemment et les données sont collectées. Elles sont alors stockées dans un fichier binaire dont le format est propre à `GPerfTools`. Les architectures supportées par `GPerfTools` sont relativement variées, puisqu'il suffit d'être capable de recompiler la bibliothèque sur l'architecture cible, ainsi que sa dépendance, la `libunwind`. Cette dernière, par exemple, ne supporte pas le Xeon Phi et empêche donc l'utilisation de `GPerfTools` sur Xeon Phi. Il est néanmoins possible de se passer de la `libunwind` en utilisant directement les fonctions disponibles dans la `libc`. Néanmoins les développeurs de `GPerfTools` recommandent de ne pas le faire pour les architectures 64 bits étant donné que cela pose des problèmes de stabilité tels que des *deadlocks*, ce qui exclut donc le Xeon Phi également. Pour pouvoir analyser les données collectées, il sera donc nécessaire de les convertir depuis le format binaire propre à `GPerfTools` vers un format lisible directement ou *via* une autre application. Un outil nommé `pprof` est fourni, qui permet de faire la conversion notamment au format `valgrind` pour pouvoir utiliser `Kcachegrind`.

On retrouve l'approche de profilage utilisant une bibliothèque avec `HPCToolkit`[6] (Adhianto et al., 2010) dont l'objectif annoncé est clairement de permettre le profilage d'applications issues du calcul haute-performance. Son but est de permettre le profilage avec un *overhead* limité (1 à 5%) tout en

[5] http://code.google.com/p/gperftool
[6] http://hpctoolkit.org/

étant capable de passer à l'échelle comme l'application qui est profilée. Il vient avec tous les outils nécessaires pour réaliser la prise de données, la mise en forme, l'affichage et l'analyse. Afin de réaliser le profilage de l'application, `HPCToolkit` va charger, comme `GPerfTools`, une bibliothèque au démarrage de l'application, en le masquant à l'utilisateur final, grâce à l'utilisation de la commande « `hpcrun` ». Pour pouvoir exploiter les données fournies par le profilage, il va d'abord être nécessaire de demander à `HPCToolkit` d'analyser l'application et les mesures effectuées *via* la commande « `hpcprof` », qui va conduire à la création d'une base de données de performances. Il sera alors possible de visualiser les résultats du profilage grâce à l'application `hpcviewer`, comme montré sur la figure 3.

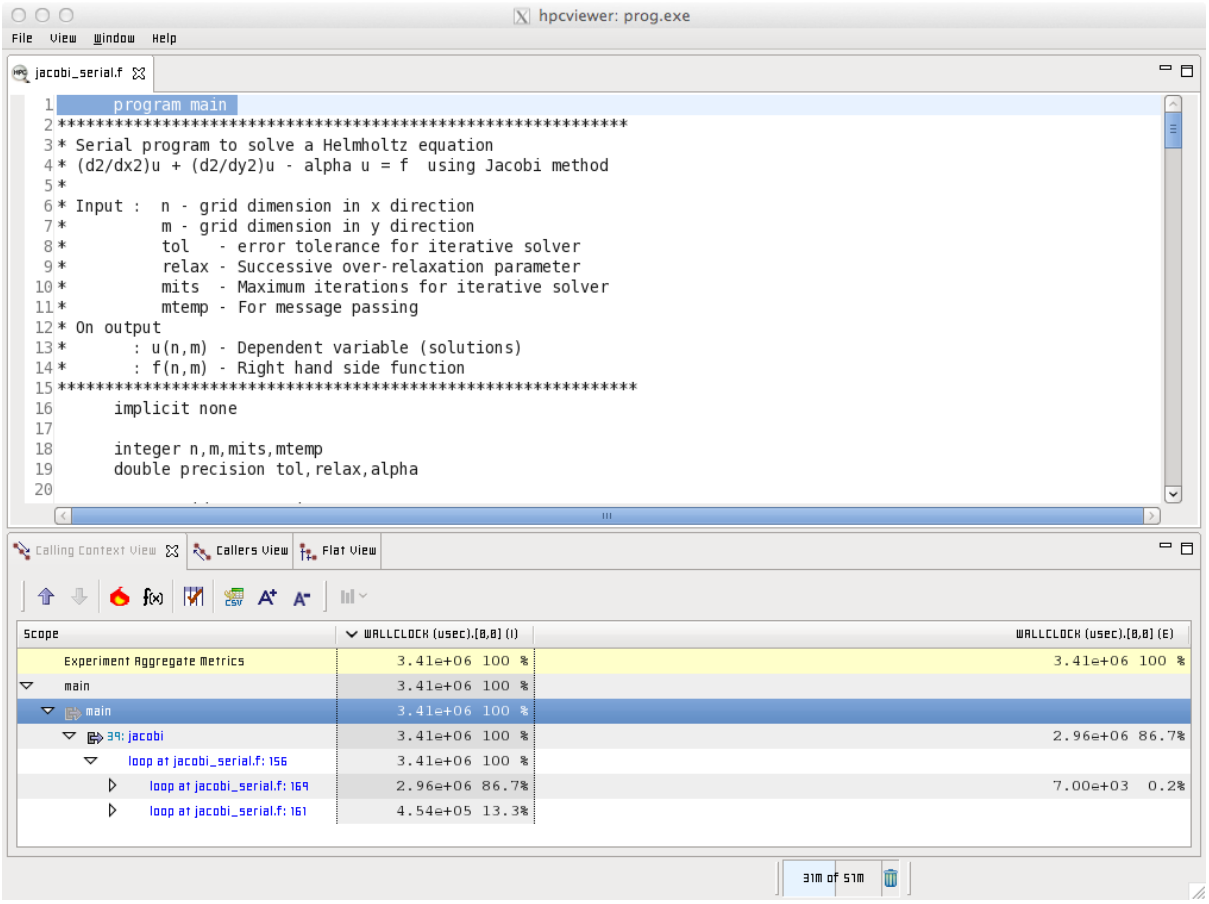

Figure 2-3 Affichage des performances d'une application dans `hpcviewer` (http://nci.org.au/services-support/training/parallel-programming/)

Sur la figure 3, on peut voir les résultats du profilage par `HPCToolkit` d'une application écrite en Fortran. Le coût analysé ici est le temps passé dans chaque boucle (et non dans chaque fonction !). Le code est également disponible pour permettre de resituer les coûts dans le code source.

Une quatrième approche de profilage est utilisée par MAQAO[7] (*Modular Assembly Quality Analyser and Optimizer*) (Djoudi et al., 2005). MAQAO peut utiliser deux méthodes pour profiler une application. La première méthode est une analyse statique de l'application. Le binaire de l'application étant donné, MAQAO va extraire directement des données de performances de l'exécutable, sans même le faire tourner, en évaluant les chemins pris et l'efficacité du code généré par le compilateur. Les métriques auxquelles s'intéressera MAQAO dans ce cas seront alors par exemple, la vectorisation du code, l'utilisation des caches L1 & L2, l'utilisation de la RAM. Il fera également une projection sur les performances dans le cas où l'application est totalement vectorisée (Barthou, Rubial, Jalby, Koliai, & Valensi, 2010). La deuxième méthode d'analyse des exécutables se fait *via* une analyse plus standard de l'application lorsqu'elle est exécutée. Néanmoins, ici aussi, la méthode repose sur une pratique peu courante. Au lieu d'attacher une bibliothèque partagée au démarrage ou bien d'émuler un environnement virtuel, MAQAO désassemble le binaire et rajoute ses instructions d'instrumentation pour ensuite re-assembler l'application et l'exécuter, et ainsi pouvoir la profiler lors de l'exécution. Cette étape s'appuie sur l'outil MADRAS (Multi Architecture Disassembler Rewriter and Assembler) (Valensi & Barthou, 2009) développé dans le même laboratoire. Parce que MAQAO s'appuie sur le binaire directement pour le profilage, il est fortement dépendent de la plate-forme cible et même du compilateur. La documentation de MAQAO demande l'utilisation du compilateur C d'Intel (ICC) et supporte les architectures 64 bits et Xeon Phi.

Une cinquième approche de profilage (sous Linux uniquement) consiste à s'appuyer directement sur le noyau Linux. Celui-ci fournit en effet plusieurs API (*Application Programming Interface*) qui permettent le profilage des applications. Historiquement, la plus ancienne est `oprofile`[8] (Levon & Elie, 2004), qui nécessite l'installation de l'outil éponyme pour être utilisée, ainsi que d'un démon. `oprofile` s'appuie sur un compteur matériel des CPU, `CPU_CLK_UNHALTED`, qui permet d'avoir le nombre de cycles consommés par un thread lorsqu'il n'est pas en attente. Par la suite, l'API `perfmon2`[9] (Eranian, 2006) a été incorporée dans Linux. Lors du développement de `perfmon2`, un premier objectif était de fournir une interface générique permettant d'accéder aux compteurs de performances matériels en étant portable quel que soit le compteur ou le constructeur. Un autre objectif était déjà le support des threads, et plus spécifiquement le support du profilage par thread. Le tout bien, évidemment, sans recompilation de l'application, juste *via* l'utilisation d'un appel système. Une des applications pouvant être utilisée pour le profilage grâce à `perfmon2` est `pfmon`. A noter qu'`oprofile` et `pfmon` ne sont pas disponibles sur Xeon Phi. Une troisième interface a été développée dans le noyau Linux, connue simplement sous le nom de « *Perf Events* » ou

[7] http://maqao.org/
[8] http://oprofile.sourceforge.net
[9] http://perfmon2.sourceforge.net/

« perfcounters ». Elle est arrivée avec le noyau 2.6.31. Cette interface a été développée pour plusieurs raisons, notamment pour éviter le besoin d'un démon supplémentaire (comme sur oprofile) et, étant intégrée dans le noyau Linux, il y avait une volonté d'intégrer un code qui soit maintenu par les développeurs de Linux (ce qui excluait donc un projet tiers). C'est la raison pour laquelle l'outil perf, nécessaire pour exploiter cette interface, est disponible avec n'importe quel noyau depuis le 2.6.31, y compris donc sur un Xeon Phi qui exploite le noyau 2.6.38. Un autre mécanisme, DTrace (McDougall, Mauro, & Gregg, 2006), venant de Solaris est disponible dans le noyau Linux. DTrace permet d'instrumenter le noyau et donc, de communiquer directement avec lui via des scripts et l'application éponyme. *Via* DTrace, il est possible de manipuler les structures du noyau, les processus ou même les fichiers. Il est donc possible de profiler une application ou même l'utilisation d'un fichier. De même, on peut par exemple vérifier les paramètres passés à chaque appel système grâce à DTrace. C'est un outil beaucoup plus générique qu'une simple interface de profilage.

A la frontière entre les approches qui nécessitent une recompilation et celles qui n'en nécessitent pas, on trouve une sixième catégorie d'approches, employée avec les langages qui disposent d'une compilation juste à temps (JIT) tels que le Java ou le Python. Ces deux langages n'étant pas disponibles sur Xeon Phi, les méthodes décrites dans la suite du paragraphe ne sont pas disponibles sur ce matériel. L'approche utilisée par Python est de fournir deux modules de profilage, profile et cProfile, qui partagent la même interface (pour faciliter l'interopérabilité). Ceux-ci peuvent être utilisés de deux façons différentes. Ou bien, l'utilisateur modifie le script Python qu'il cherche à optimiser et dans la fonction main, il invoque la méthode run du profiler, ou bien, *via* la ligne de commande Python, il charge le profiler, comme montré dans le code 2.

```
Python –m cProfile [-o fichier] script.py
```

Code 2-2 : Utilisation du module cProfile pour profiler un script Python

Par défaut, les données du profilage seront affichées sur la sortie standard. Il est néanmoins possible de les écrire dans un fichier, comme montré dans le code 2 grâce à l'option « -o ». Le profilage *via* cProfile donne les informations par fonction : nombre d'appels, temps total, temps par appel, temps cumulé (temps de la fonction et de ses filles), temps cumulé par appel.

2.3 – Quelques outils de profilage d'une application avec recompilation

Il existe d'autres outils pour profiler une application, ceux-ci pouvant recourir explicitement à la recompilation de l'application pour y parvenir.

Un premier outil pour ce profilage est TAU[10] (*Tuning and Analysis Utilities*) (Shende et al., 1998). En réalité, cet outil peut également être utilisé sans recompilation préalable, mais dans un tel cas, il ne réalisera le profilage que des appels de fonctions pour MPI (*Message Passing Interface*) (Snir, Otto, Walker, Dongarra, & Huss-Lederman, 1995) ce qui limitera la capacité de profilage de l'application en intégralité. Pour le profilage, TAU s'appuie notamment sur la bibliothèque PAPI[11] (*Performance Application Programming Interface*) (Mucci, Browne, Deane, & Ho, 1999). Cette bibliothèque sert de couche d'abstraction du matériel. En effet, les compteurs de performance matérielle ne sont pas standardisés et chaque vendeur est libre de son implémentation. PAPI abstrait donc ces compteurs et expose une API standardisée quel que soit le matériel, à l'image de ce qui est fait avec perfmon2. Par simplification, elle sélectionne également les compteurs qui ont le plus de sens et ignore les autres. Néanmoins, en cas de besoin pour l'implémenteur, il est possible de requêter directement les compteurs matériels. TAU supporte de nombreuses architectures utilisées dans le domaine du calcul à haute performance y compris le Xeon Phi.

Lors de l'utilisation de TAU, un ou plusieurs fichiers binaires seront créés. Il faut alors utiliser un autre outil pour pouvoir les visualiser et les analyser. TAU vient avec l'outil `ParaProf` qui permet de visualiser les données de performance, comme montré sur la figure 4, où ont été représentées les performances des appels MPI sur les différents nœuds. Il vient également avec l'outil `JumpShot` (Zaki, Lusk, Gropp, & Swider, 1999) qui permet de représenter les traces de communications MPI notamment, ce qui est montré sur la figure 5, où l'on voit les communications MPI dans le temps.

[10] https://www.cs.uoregon.edu/research/tau/home.php
[11] http://icl.cs.utk.edu/papi/

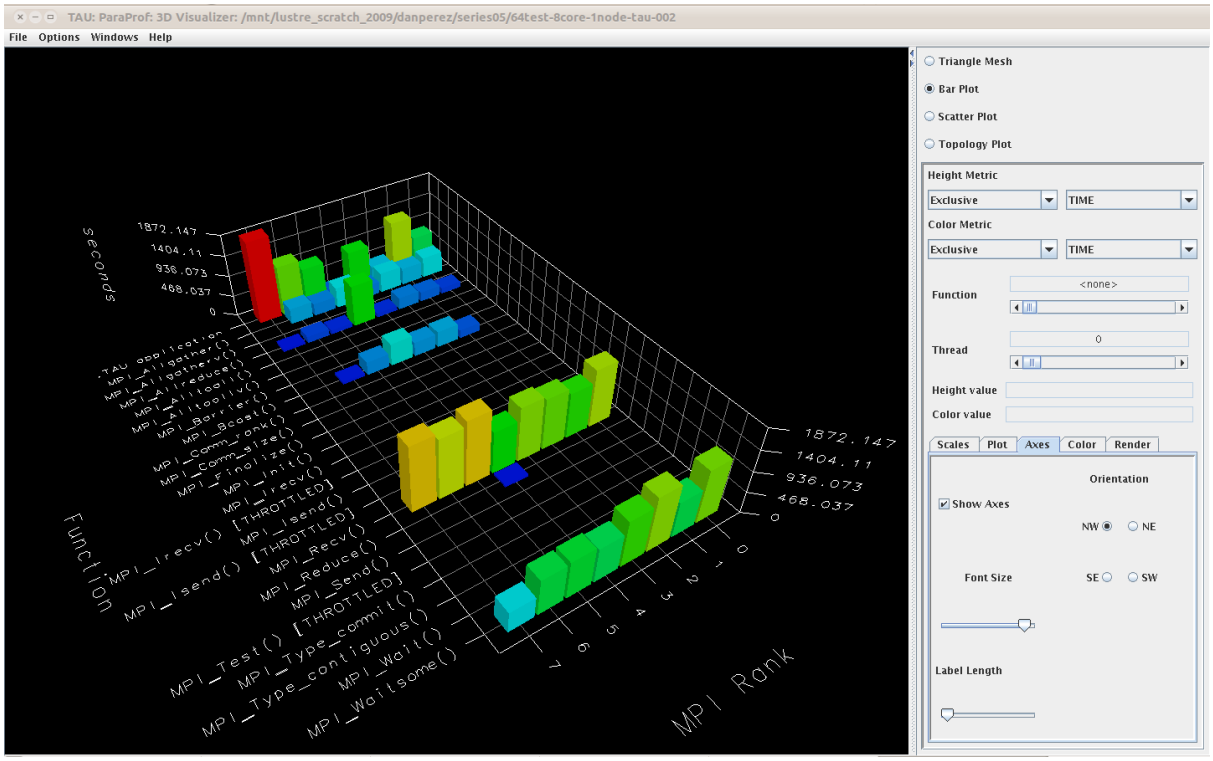

Figure 2-4 Affichage des performances des appels MPI dans `ParaProf`

(https://wiki.hpcc.msu.edu/download/attachments/13863151/tau-paraprof-3d.png)

Dans les outils spécialisés pour l'analyse des applications MPI ou OpenMP, on retrouve également VAMPIR[12] (*Visualization and Analysis of MPI Resources*) (Nagel, Arnold, Weber, Hoppe, & Solchenbach, 1996) ou encore `Scalasca`[13] (F. Wolf et al., 2008). Une importante différence entre les deux réside dans une fonctionnalité assez particulière de `Scalasca`. En effet, ce dernier analyse automatiquement les données issues du profilage pour n'en exposer à l'utilisateur final que les parties intéressantes sur lesquelles un travail peut apporter quelque chose. Alors que VAMPIR utilise sa propre interface pour afficher toutes les données issues du profilage, visibles sur la figure 6, `ParaProf` peut être utilisé pour afficher les données de `Scalasca`. Pour le profilage, VAMPIR et `Scalasca` s'appuient sur le cadriciel `Score-P`[14] (qui peut s'appuyer sur TAU pour l'instrumentation ou sur OPARI pour le code OpenMP (Mohr, Malony, Shende, & Wolf, 2002)) (Knüpfer et al., 2012). Score-P requiert donc de recompiler l'application avant de pouvoir la profiler. A noter que VAMPIR ne nécessite pas explicitement `Score-P` pour le profilage, même s'il est fortement recommandé par les développeurs de VAMPIR de l'utiliser. Il est également possible d'utiliser un ancien logiciel, qui fait maintenant partie d'OpenMPI, `VampirTrace`, mais ce dernier

[12] https://www.vampir.eu/
[13] http://www.scalasca.org/
[14] http://www.vi-hps.org/projects/score-p/

est aujourd'hui abandonné. Au moment de la rédaction de ce manuscrit, aucun de ces logiciels ne supporte le profilage d'application sur Xeon Phi. Deux autres outils existent également pour le profilage d'applications OpenMP : `ompP` (Fürlinger & Gerndt, 2008) ou MPI : `mpip` (Vetter & Chambreau, 2005).

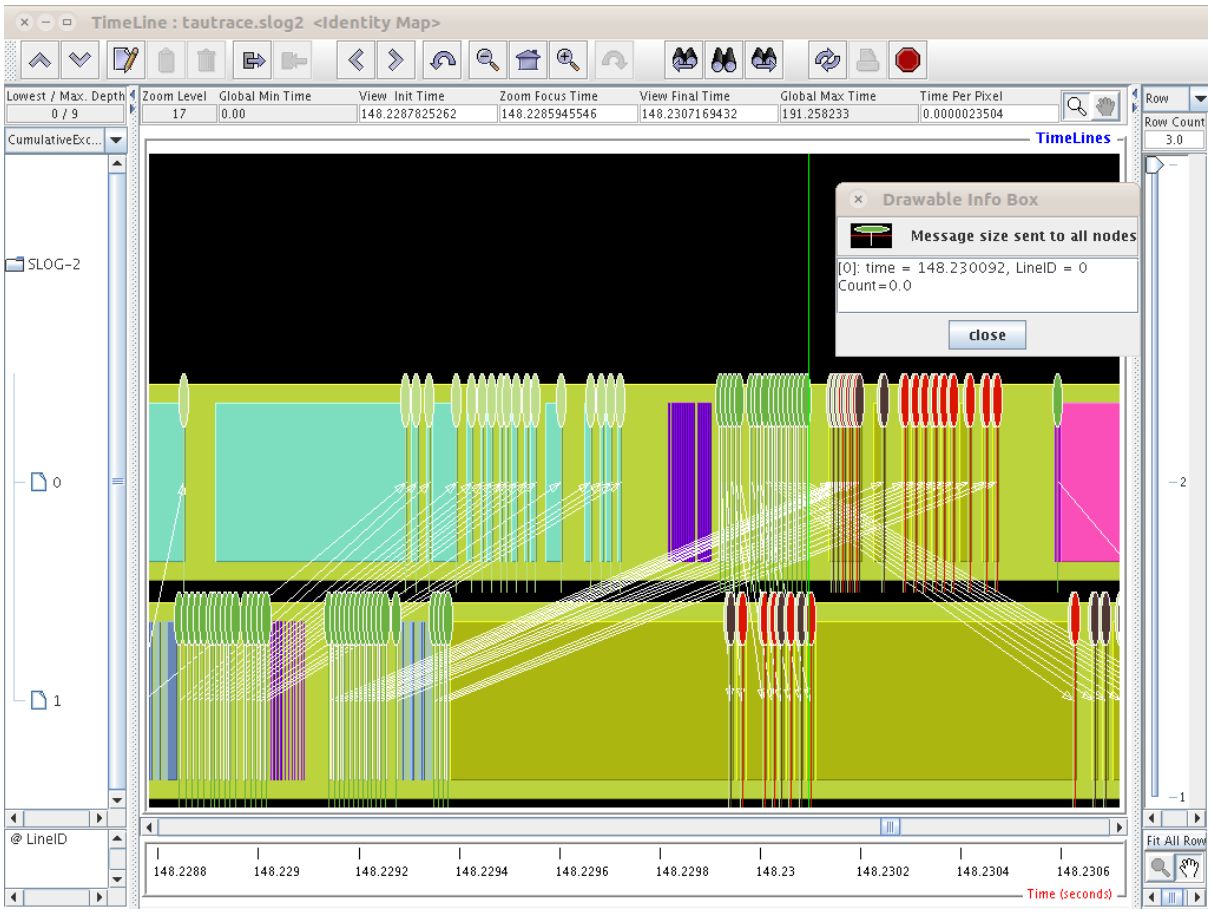

Figure 2-5 Affichage des communications MPI dans le temps dans `JumpShot` (https://wiki.hpcc.msu.edu/download/attachments/13863151/tau-jumphsot-zoom.png)

Sur la Figure 6, on peut clairement voir le lancement des différents processus ainsi que leurs communications (marquées par les traits noirs entre les processus). Par ailleurs, les appels bloquants MPI (`MPI_Wait`) sont marqués en rouge sur la figure. Plus leur nombre et leur durée sont réduits, plus le programme sera performant. Ce profilage se fait au niveau des fonctions et non au niveau des instructions.

Enfin, on trouve des outils spécialisés dans le profilage, non pas des performances directes en terme de calcul, mais par exemple dans le profilage de l'utilisation des caches disponibles sur le nœud de calcul, de la même façon que `cachegrind`. Par exemple, il existe l'ensemble d'outils CACHEVIZ

(*CACHE behavior VIsualiZer*) (Yu, Beyls, & D'Hollander, 2001) qui permet de simuler l'utilisation des caches dans une application. Un premier outil va permettre d'instrumenter le code de l'application pour pouvoir la recompiler en activant le profilage. Dès lors, à la prochaine exécution, l'utilisation des caches sera profilée et rapportée dans un fichier de sortie. Ensuite, un outil permet d'ouvrir ce fichier de sortie pour visualiser l'utilisation des caches par l'application.

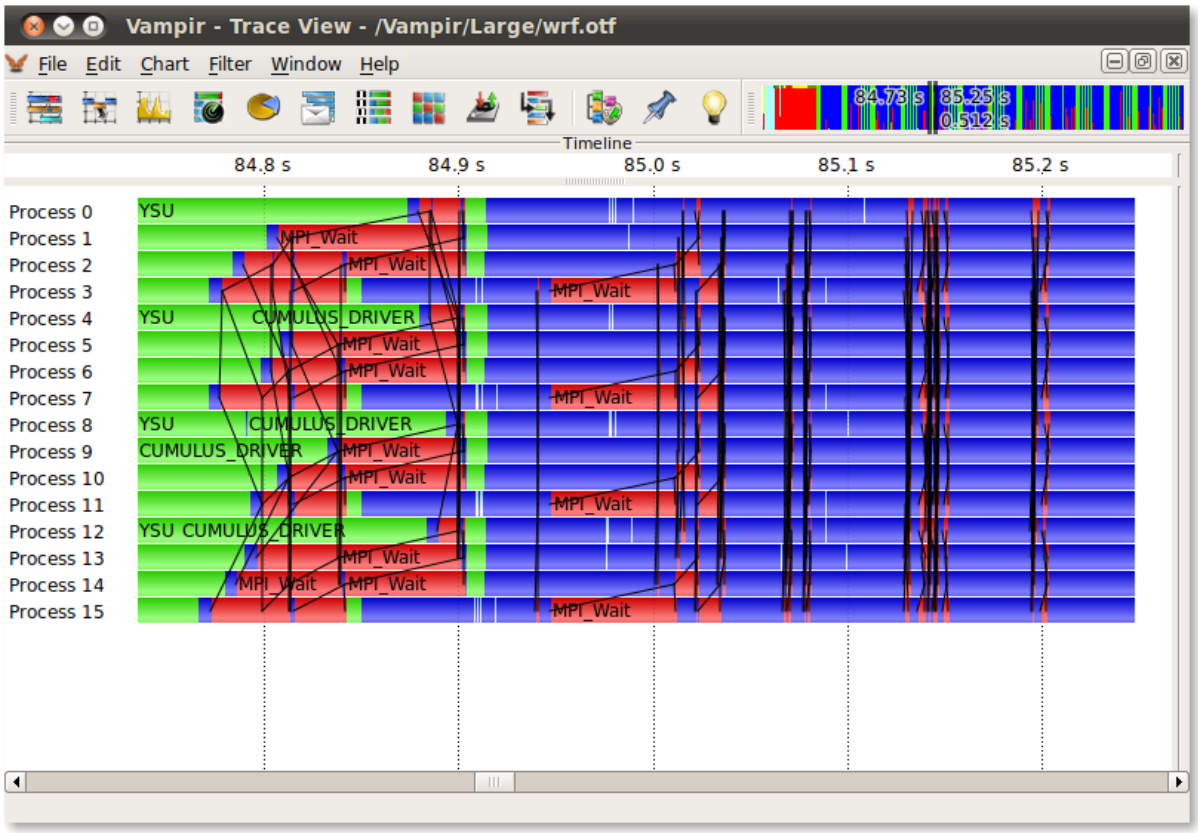

Figure 2-6 Affichage des processus MPI et de leurs communications avec VAMPIR (https://www.vampir.eu/tutorial/manual/performance_data_visualization)

On constate dans la littérature qu'en pratique, pour le profilage, généralement un seul outil ne suffit pas pour avoir des données satisfaisantes. Ainsi dans (Martin-Haugh, 2013), on constate que perf et `valgrind` ont été utilisés. Il est également fait référence à GOoDA (*Generic Optimization Data Analyzer*) (Calafiura, Eranian, Levinthal, Kama, & Vitillo, 2012). L'objectif de GOoDA est de simplifier, pour l'utilisateur, les problématiques liées aux cycles utilisés dans un processeur en lui permettant de savoir comment ils ont été utilisés sans avoir besoin d'une connaissance approfondie sur l'architecture matérielle. Dans (Nowak, 2010), on retrouve encore plus de logiciels utilisés : `valgrind`, `GPerfTools`, `perfmon2`, `VTune`, ainsi que des outils développés en interne. Dans

(Chauhan et al., 2013), on retrouve l'usage de PAPI et de `GPerfTools`, mais il est également fait mention de `Pin`[15] (Luk et al., 2005). `Pin` est un outil développé par Intel qui permet le profilage d'une application en insérant du code d'instrumentation dans l'application à profiler à « chaud », sans recompilation préalable. C'est la raison pour laquelle d'ailleurs, dans (Chauhan et al., 2013), le passage de PAPI à `Pin` a été réalisé, pour éviter la recompilation nécessaire.

Une autre approche pour le profilage d'application avec recompilation peut s'appuyer sur le paradigme de Programmation Orientée Aspect (Kiczales et al., 1997). Ce paradigme de programmation est totalement découplé des paradigmes déjà existants : programmation orientée objets (Rentsch, 1982), programmation impérative, programmation fonctionnelle (Henderson, 1980). Il peut s'appliquer à chacun d'entre eux ; il s'agit d'un paradigme transverse. La programmation orientée aspect (AOP – *Aspect Oriented Programming*) consiste à venir « greffer » du code supplémentaire dans du code principal déjà existant, de façon transverse. Cela permet par exemple, dans le cadre d'un développement d'un applicatif de séparer la logique métier de la logique technique. La logique métier sera le cœur de l'applicatif, qui traitera les données et produira les résultats. La logique technique sera toute la « décoration autour » : quelles contraintes sur les données ? Comment mettre en forme les données ? Comment les récupérer ? etc. Autant la logique métier est supposée rester constante, autant la logique technique peut être amenée à prendre plusieurs formes en même temps selon l'utilisateur à qui on s'adresse, et à évoluer dans le temps. L'objectif de l'AOP est donc de rendre les applications le plus modulaire possible. Ici, les modules seront des aspects, qui viendront s'ajouter au programme de façon transverse. Ceux-ci peuvent également être dénommés « *advice* ». L'endroit où sont appliqués les aspects est géré grâce aux « *pointcuts* », les points de coupure. Ceux-ci peuvent, par exemple, correspondre à des fonctions. La position de ces points de coupure est gérée grâce à des indications « *before* », « *after* », « *around* ». Dans le cadre de la fonction, cela permet donc de savoir si l'aspect est ajouté avant la fonction, après, ou s'il englobe la fonction (« *around* » pouvant donc être vu comme la somme de « *before* » et de « *after* »). Un exemple concret d'utilisation peut être la gestion des pointeurs dans un langage tel que le C. En C, on ne tolère pas les pointeurs de valeur 0, dits pointeurs nuls, leur déréférencement étant quelque peu problématique. On peut donc ne pas vérifier la valeur des pointeurs dans la logique métier qui n'est là que pour traiter les données, et rien de plus, et déléguer cela à un aspect, qui vérifiera, par exemple, que chaque fonction appelée ne reçoit pas un pointeur nul. Selon que l'on soit en production ou en développement, on pourra avoir, respectivement, une dépendance sur un aspect qui n'affichera que la pile d'exécution jusqu'à l'appel, ou bien jusqu'à l'appel à un débogueur, avec plus d'informations affichées. On trouve plusieurs autres exemples plus

[15] https://software.intel.com/en-us/articles/pin-a-binary-instrumentation-tool-downloads

conséquents dans (Coady & Kiczales, 2003) qui cite des usages de programmation orientée aspect dans le cadre du développement d'un système d'exploitation, et plus particulièrement, de son noyau. L'argument de modularité est important dans ce dernier cas, afin de pouvoir moduler les fonctionnalités du noyau en fonction des souhaits de l'utilisateur.

Même si ici, on a pris l'exemple du C, l'AOP étant un paradigme de programmation, il est indépendant du langage de programmation. Certains langages ont de manière native des fonctionnalités plus ou moins évoluées pour faire du développement orienté aspect. Le langage Python dispose par exemple de « décorateurs ». Le langage Java, quant à lui, dispose d'annotations, dont les capacités sont étendues à chaque nouvelle version. Néanmoins celles-ci n'offrant pas toutes les fonctionnalités du paradigme, des cadriciels supplémentaires tels que `AspectJ` (Gradecki & Lesiecki, 2003) ont été développés pour permettre une utilisation complète du paradigme en Java. Dans le cas du C et du C++, l'AOP n'est pas supporté nativement par le langage, et il faut donc des cadriciels qui permettent d'étendre le langage. Pour le C, il existe `AspectC` ("AspectC: AOP for C," 2004) qui s'appuie sur les concepts de `AspectJ`. Pour le C++, deux cadriciels spécifiques existent : `FeaturesC++` (Apel, Leich, Rosenmüller, & Saake, 2005) et `AspectC++` (Spinczyk, Gal, & Schröder-Preikschat, 2002). Le premier ne fournit pas exactement toutes les fonctionnalités du paradigme AOP. Il permet surtout de palier plusieurs manques du C++ : il permet des déclarations de classes partielles, qui seront complétées (et spécialisées) dans la suite du programme. De la même façon, il permet d'étendre à la volée des classes déjà implémentées. Enfin, il fournit le mot clé « `super` », qui comme en Java permet de se référer à la classe mère à partir d'une classe fille. `AspectC++` lui est une véritable implémentation du paradigme orienté aspect. Il fournit les mécanismes nécessaires et étend le langage C++ avec des mots clés propres (tels que « `aspect` ») pour pouvoir implémenter des aspects. Il se place au niveau de la fonction (qu'elle soit membre d'une classe ou non) pour les points de coupure. Dans `AspectC++`, il est possible de découper un unique aspect en plusieurs morceaux, qui sont nommés « `advice` », chacun pouvant avoir ses propres points de coupure. L'application des `advice` (et leur choix) se fait à la compilation. Cette étape est nommée l'étape de tissage (ou de *weaving*) où le code des différents aspects est inséré dans le code métier aux différents points de coupure. Le code C++ résultant est alors transmis au compilateur C++ pour que la compilation soit effectuée.

Il est donc clairement possible de pouvoir réaliser un profileur (ou également un débogueur) avec la programmation orientée aspect. Il est possible de profiler les fonctions grâce à un aspect et ainsi de suivre la trace d'exécution et les coûts associés. C'est quelque chose qui a été fait en Java, grâce à `AspectJ` dans `DJProf` (Pearce, Webster, Berry, & Kelly, 2007), et sans `AspectJ` dans (Ansaloni, Binder, Villazón, & Moret, 2010; Binder, Ansaloni, Villazón, & Moret, 2011).

Une fois les données de performances acquises (quelle que soit la méthode), elles sont analysées afin d'identifier les points chauds où une optimisation est nécessaire. Ces points chauds sont les fonctions ou les portions de code ayant un coût élevé, qui sont fréquemment appelés, ou bien qui représentent une partie majeure du programme. Les zones à cibler en priorité pour les optimisations sont bien évidemment les points chauds : ce sont eux qui apporteront le maximum de gain de performance.

3 – Optimisation d'applications

L'objectif principal d'une optimisation est de réduire au moins l'un des coûts qui a été mesuré. Néanmoins, cela ne veut pas dire réduire tous les coûts associés à une portion de code. En effet, il peut être nécessaire de déborder sur un coût B pour optimiser un coût A. Par exemple, pour optimiser la consommation mémoire d'une portion de code, on peut imaginer qu'on recalculera à chaque fois qu'elles sont nécessaires une partie des données sur lesquelles on travaille. Cette méthode aura pour effet une augmentation sensible de la consommation processeur de la fonction. Inversement, on pourra imaginer stocker les données calculées en mémoire pour les avoir directement dès que nécessaire. Cette méthode aura pour effet d'augmenter la quantité de mémoire requise pour le fonctionnement du code. L'optimisation dépend donc réellement du coût à réduire, de l'architecture et de la plate-forme sur laquelle on travaille.

3.1 – Du bon usage du processeur

Par défaut, un compilateur n'optimise pas le binaire qu'il génère pour un processeur particulier. Il reste toujours avec le minimum d'optimisations possibles, pour que l'exécutable produit puisse tourner sur n'importe quelle machine de la même architecture, même si il lui manque des jeux d'instructions dont disposent les tous derniers processeurs. Ceci a pour conséquence que les nouvelles instructions ajoutées dans les nouveaux processeurs sont ignorées, alors qu'elles pourraient permettre de gagner en performance. Il est donc nécessaire d'indiquer au processeur sur quelle architecture et surtout sur quelle génération de cette architecture l'application va être exécutée, afin qu'il puisse utiliser (s'il les supporte) les jeux d'instructions les plus récents. Par exemple, si on compte exécuter l'application sur la machine où elle est compilée, un simple « `-march=native` » pour GCC ou « `-xHost` » pour ICC fera utiliser le plus haut jeu d'instructions disponible. Bien évidemment, il sera encore nécessaire de dire au compilateur d'utiliser un

maximum d'optimisations possibles, grâce à l'option « `-O2` » qui optimise l'application en performance. Si on veut un exécutable réduit en empreinte mémoire, il existe aussi l'option « `-Os` ». Il est à noter que l'usage de l'option « `-O3` » peut conduire à des problèmes de performances (Intel, 2010) voire à la révélation de bugs. En effet, cette option active d'autres optimisations souvent classées sous le nom d' « optimisations agressives » dont les effets ne sont pas toujours bénéfiques. On peut se retrouver à dépasser la pile d'une application, ou à sortir du cache alors que ces dépassements n'avaient pas lieu avant l'optimisation. Ou encore, sur des boucles particulières, on peut avoir des dépassements sur les zones de mémoire tampon. Il faut donc être particulièrement prudent quant à son usage. Il est préférable donc de compiler son programme deux fois, une fois avec « `-O2` », puis l'autre avec « `-O3` », puis de comparer le fonctionnement des deux versions. Dans les cas où ils retournent tous les deux le même résultat (correct), on peut prendre celui qui a les meilleures performances.

3.1.1 – Optimisations initiales

Avant de songer à aller plus loin dans les optimisations fortes du programme, il convient d'utiliser quelques optimisations de base dans l'écriture du code. Elles permettront au compilateur de simplifier le code généré. Celles-ci sont plus de l'ordre de la bonne pratique de développement, qui conduit à de meilleures performances. Dans le cadre d'un code en C/C++, que ce soit pour Xeon Phi ou pour un processeur classique, le simple fait d'utiliser le mot clé « `const` » permet de gagner en performances. En effet, ce mot clé permet de signifier les valeurs stockées qui resteront constantes durant les calculs et qui ne seront donc pas modifiées. Dès lors, le compilateur peut optimiser le programme. Plutôt que de récupérer la valeur à chaque fois (ou pire, de la recalculer à chaque fois), le compilateur peut la calculer une unique fois et la remplacer partout dans le code par sa valeur ; il s'agit de la technique d'optimisation connue sous le nom de « propagation des constantes ». A noter que le calcul unique de la constante à la compilation pour ne pas avoir à le faire à l'exécution, quand cela est possible, est connu sous le nom de « évaluation des constantes ». Dans des cas extrêmes, tel que celui présenté dans (Vladimirov & Karpusenko, 2013), avec ce simple changement, les auteurs ont pu constater un gain de performances de 4,5X sur leur application de test.

De la même façon, il est conseillé de ne pas utiliser de pointeurs, ni leur arithmétique, pour accéder aux membres d'un tableau. L'utilisation de simples index permet au compilateur d'optimiser le code plus fortement, notamment de vectoriser si cela est possible (cf. : section 3.1.2). Pour un cas

applicatif profitant de ce style d'écriture et d'un processeur avec des capacités vectorielles, les auteurs de (Vladimirov & Karpusenko, 2013) ont constaté un gain de performances de 5X.

Dans le cadre des boucles, une optimisation importante peut être de définir correctement la condition de sortie de la boucle. Si on veut faire une boucle sur 42 éléments, intuitivement, on écrira une boucle type « `for` » avec une variable initialisée à 0 et que l'on incrémentera de 1 en vérifiant qu'elle est inférieure strictement à 42. Néanmoins, comparer la valeur de deux variables pour un processeur est une opération délicate : il va d'abord faire la soustraction des deux valeurs, puis vérifier si la différence est nulle. On se retrouve dès lors avec deux opérations. Il est donc plus judicieux d'avoir une variable qui part de 41 et qui est décrémentée de 1. On se retrouve alors comme condition de sortie à comparer la valeur de la variable à 0, ce qui est bien plus efficace pour le processeur.

Il est également important d'éviter la duplication inutile de calculs, notamment d'appels de fonctions. Il est peut donc être important d'utiliser des variables intermédiaires pour stocker les résultats et ainsi éviter un nouveau calcul. Ce type de duplication peut se retrouver dans des cas inattendus. Notamment dans le cas de boucles imbriquées où un calcul n'est dépendant que de la première boucle, mais utilisé dans la seconde. Il est inutile de le calculer à chaque itération de la seconde boucle. Il suffit de le calculer dans la première et de stocker le résultat ; il s'agit de la technique d'optimisation connue sous le nom de « suppression des invariants de boucles ». De la même façon, l'utilisation des macros peut conduire à la duplication de calculs. En effet, la macro est remplacée à la compilation par le code qu'elle contient. Dès lors, si un de ses arguments est un appel de fonction et que cet argument est utilisé plusieurs, alors, il y a aura duplication de l'appel. Le code 3 permet d'illustrer ce propos. En effet, en première approche, si « `my_min` » était une fonction, il n'y aurait bien qu'un seul appel à « `my_funct` » pour a et pour b. Néanmoins, « `my_min` » étant une macro, à la compilation, son appel sera directement remplacé par le code. On se retrouvera donc avec trois appels : d'abord pour déterminer le minimum entre `my_funct(a)` et `my_funct(b)`, et ensuite un troisième pour retourner le résultat. Il convient donc, ici aussi, d'utiliser des variables intermédiaires pour les deux premiers appels à `my_funct`, afin d'éviter une redondance d'appels.

```
#define my_min(a, b) ((a > b) ? b : a)
auto min = my_min(my_funct(a), my_funct(b));
```

Code 2-3 : Calcul du minimum entre deux valeurs *via* une macro

Enfin, il est important d'éviter l'utilisation de la division en virgule flottante. Celle-ci est plus coûteuse que la multiplication (Vladimirov & Karpusenko, 2013). Dès lors, au lieu de diviser par une constante, il conviendra de multiplier par son inverse. Dans le cas où il s'agit de division de variables, il faudra s'assurer de minimiser le nombre de divisions.

De la même façon, certaines optimisations n'ont pas à être faites par le développeur. Elles vont généralement être réalisées automatiquement par le compilateur, parce qu'elles sont totalement anodines d'un point de vue du résultat, mais ont un impact réel sur le temps d'exécution. Généralement, elles ne seraient pas faites par le développeur, parce qu'elles nuiraient à la qualité du code ou à sa maintenabilité. Ainsi, pour les calculs dans le code, le compilateur emploie la technique dite d'« optimisation algébrique ». Celle-ci consiste à remplacer des petits bouts de calcul par des calculs plus rapide. Par exemple, une élévation au carré, qui est une opération lourde, sera remplacée par la multiplication du nombre par lui-même. De même, la multiplication par une puissance de deux d'un nombre entier sera remplacée par un décalage de bits vers la gauche. La division par une puissance de deux d'un nombre entier sera, quant à elle, remplacée par un décalage de bits vers la droite. Le compilateur pourra aussi utiliser la technique d'« élimination des sous-expressions communes ». Celle-ci consiste à stocker, dans des variables intermédiaires, des résultats de traitements sans effets de bords sur les données qui seraient réalisés plusieurs fois dans le code, alors qu'ils peuvent être réutilisés directement. Une autre optimisation réalisée par le compilateur est la technique dite d'« élimination de code mort ». Le compilateur va purement et simplement supprimer le code qui ne sera jamais atteint. Soit parce qu'il y a des conditions booléennes qui ne seront jamais atteintes, soit parce qu'il y a des retours avant ce code, soit parce qu'il y a des calculs qui ne seront jamais utilisés mais qui sont pourtant réalisés. Généralement, les compilateurs récents suppriment ce code mort en affichant un « *warning* » à l'utilisateur. Un tel code mort est dans la majorité des cas révélateur d'un problème de conception dans le code : mauvaise condition booléenne, variable inutilisée, etc. Dans les

optimisations de boucle, le compilateur va potentiellement réaliser la technique nommée « déroulage de boucles » (très connue sous le nom anglo-saxon « *loop unrolling* ») : si le développeur utilise une boucle pour traiter un ensemble (restreint) de données, le compilateur supprime la boucle et duplique le code pour réaliser le traitement. Cela allège l'exécution, en évitant de maintenir une ou plusieurs variables d'état de la boucle et de faire des comparaisons sur la condition de sortie de la boucle. Cette optimisation n'est cependant possible de façon complète que si le compilateur a une possibilité d'évaluer la limite de la boucle. Dans les autres cas, le compilateur peut dérouler partiellement une boucle. S'il sait par exemple que le traitement sera toujours fait un nombre pair de fois, il peut dupliquer deux fois le code, et effectuer deux traitements à chaque itération de la boucle.

3.1.2 – Exploitation des instructions vectorielles

Activer la majorité des optimisations ne permettra pas forcément au compilateur d'exploiter au mieux l'architecture pour laquelle est compilé le programme. En effet, pour certaines optimisations, notamment la vectorisation, le compilateur doit avoir certains prérequis dans le code pour éviter d'introduire des bugs (notamment des hypothèses d'indépendance des données). Dans certains cas, il ne peut pas arriver à la conclusion que les prérequis sont atteints, soit parce que l'écriture même du code est trop complexe, soit parce que les données sont adressées à plusieurs endroits et qu'il ne peut pas suivre leur utilisation correcte ou non, par exemple, identifier s'il y a recouvrement à partir d'un moment. Il convient alors d'aider le compilateur, de lui indiquer si ces conditions sont remplies ou non.

Comme indiqué dans (Sabahi, 2012), pour que le compilateur puisse vectoriser une boucle, il faut que le nombre d'itérations soit connu avant le début de la boucle (à l'exécution ou à la compilation). Il ne faut donc pas que la fin de boucle dépende des données. Ce qui implique donc que des instructions « `break` » ou « `continue` » qui dépendent des données dans la boucle sont à proscrire. Par ailleurs, tout appel à une fonction que le compilateur n'a pas à sa disposition en format vectorisé (à commencer par celles de l'utilisateur) rendra la vectorisation impossible. Le compilateur ne dispose que de quelques fonctions de « base » mathématiques en format vectorisé. Les instructions conditionnelles peuvent également poser problèmes et donc empêcher la vectorisation, si elles ne sont pas simplifiables par le compilateur. Enfin, les boucles imbriquées, si elles ne sont pas supprimables par le compilateur empêcheront la vectorisation.

Toujours d'après (Sabahi, 2012), d'autres obstacles peuvent empêcher la vectorisation. On citera notamment les accès mémoires non coalescents ou encore la dépendance entre les données. Un exemple évident de dépendance entre données est par exemple l'accès à l'élément mémoire précédent pour travailler sur le suivant. Si, en séquentiel, cela ne pose aucun problème, en vectoriel, comme plusieurs éléments sont traités à la fois, cela n'est plus le cas. Cette dépendance peut néanmoins être moins explicite, si par exemple la boucle travaille sur deux tableaux de données, l'un pointant sur une partie de l'autre. Dans ce cas aussi, le compilateur refusera la vectorisation de peur qu'il y ait une dépendance entre les données. Dans ce cas, il est alors possible d'indiquer au compilateur s'il n'y a pas de dépendance entre les données. Au début de la boucle pour laquelle le compilateur « pense » qu'il y a une dépendance, l'ajout du `pragma` « `#pragma ivdep` » permettra de lui indiquer qu'il n'y a pas de dépendance et que la boucle peut être vectorisée (si c'était le seul problème).

Par ailleurs, il convient également de ne pas tenter d'être plus efficace que le compilateur. On pourra citer (Kernighan & Plauger, 1978) : « *Trying to outsmart a compiler defeats much of the purpose of using one.* ». En effet, le compilateur est l'élément, dans la chaîne de développement du logiciel, qui connaît le mieux l'architecture sur laquelle le code va être exécuté, et c'est également celui qui aura la lecture la plus fine du code. A vouloir trop optimiser, on finira par casser les schémas que le compilateur connaît et on l'empêchera d'optimiser. Par exemple, il ne faut pas tenter d'optimiser des fonctions telles que `memcpy()` ou `memset()` par des implémentations savantes à la main ; elles seront moins performantes que le code produit par le compilateur (qui ne fera même plus apparaître les appels).

Quand il s'agit d'exploiter finement les instructions vectorielles du processeur, il n'existe que peu d'alternatives. Une solution, la plus immédiate, est de réécrire le code critique, directement en assembleur en exploitant le jeu d'instructions visé. La contrepartie est qu'il faut soit se restreindre à un seul type d'instructions et ne supporter que celles-ci (par exemple SSE4, ou AVX) soit supporter plusieurs jeux d'instructions et réécrire le code en autant de versions qu'il y a de jeux d'instructions supportés. Il est heureusement possible de s'adapter au jeu d'instruction de manière dynamique, à l'exécution, un binaire pouvant inclure tous les jeux d'instructions possibles et n'utiliser que le code qui correspond au processeur sur lequel il s'exécute. A titre d'exemple, on peut citer le logiciel `FFmpeg` (Tomar, 2006) qui est utilisé pour le traitement vidéo et audio (codage / décodage). Il

s'appuie sur des instructions vectorielles pour des raisons de performances et son code est adapté pour supporter plusieurs types de jeux d'instructions possibles[16].

Une alternative est d'utiliser des bibliothèques prévues à cet effet. Des bibliothèques de fonctions telles que `Boost SIMD` (Estérie, Gaunard, Falcou, Lapresté, & Rozoy, 2014) sont prévues pour permettre l'écriture de code vectoriel dans un langage de haut-niveau. Dans le cas de `Boost SIMD`, en raison de l'utilisation de la méta-programmation grâce aux *templates* du langage C++ (Abrahams & Gurtovoy, 2004), il n'est pas nécessaire d'avoir recours à une étape de compilation intermédiaire. Un DSL (*Domain Specific Language*) est « compilé » lors de la compilation du programme, *via* les *templates*. Ce DSL permet d'exprimer la vectorisation de l'application. Ceci a l'inconvénient de fortement ralentir le processus de compilation de l'application puisque le compilateur doit d'abord évaluer les *templates* puis les intégrer dans le code source qu'il compilera enfin. Le code 4 montre un exemple d'utilisation de `Boost SIMD` en C++11. On y trouve la définition d'un vecteur de données (`rgb`) et le calcul du niveau de gris en exploitant les trois composantes dudit vecteur, le tout, de façon vectorielle.

```
auto rgb = fusion::make_vector(red, green, blue);
for (std::size_t i = 0; i != height * width;
     i += pack<float>::static_size)
{
    auto p = load<decltype(rgb)>(rgb, i);
    auto res = 0.3f * fusion::at_c<0>(p) + 0.59f *
               fusion::at_c<1>(p) + 0.11f * fusion::at_c<2>(p);
    store(res, result, i);
}
```

Code 2-4 : Utilisation de `Boost SIMD` pour faire une conversion RGB vers des niveaux de gris (http://meetingcpp.com/tl_files/mcpp/slides/12/simd.pdf)

Dans (Estérie et al., 2014) les auteurs réalisent une étude de performances de leur bibliothèque, notamment en la confrontant aux vectorisations automatiques des compilateurs. Pour leur étude de performances, ils s'appuient sur les vectorisations de GCC et de ICC. Ils ont réalisés leurs tests sur deux processeurs de deux générations différentes Nehalem et Sandy Bridge, afin de pouvoir

16 http://git.videolan.org/?p=ffmpeg.git;a=blob;f=libavutil/x86/x86util.asm;h=d6702c1466fd147b991b1d70eecef150564de9de;hb=HEAD

évaluer les performances pour les instructions SSE4.2 (Nehalem) et AVX (Sandy Bridge). Dans un premier temps, ils ont évalué les performances sur un algorithme simple qui réalise un produit scalaire sur deux vecteurs et ajoute le résultat à un troisième vecteur. Sur cet exemple, c'est en double précision que les performances de Boost SIMD ont été les moins bonnes. En prenant cette implémentation pour référence, les auteurs ont pu obtenir des gains de performances entre de 1,6X et 1,8X pour l'auto vectorisation avec GCC sur SSE4.2 et sur AVX. En utilisant ICC, ils ont pu obtenir des gains de performance entre 2,75X et 3,75X sur SSE4.2 et entre 2,5X et 3,25X sur AVX. Les performances des compilateurs étaient donc largement meilleures ici. Néanmoins, ce résultat est à mettre en perspective : c'est un exemple simple que les compilateurs savent reconnaître et optimiser plus fortement que du code directement écrit en instructions vectorielles qui fait la même chose. D'autres optimisations telles que le déroulage de boucles viennent également, ici, aider le compilateur, ce qui ne peut pas être fait avec les instructions vectorielles. Par ailleurs, les auteurs en ont profité pour montrer que l'écriture du même code vectoriel à la main donne des performances moindres que celui écrit par Boost SIMD. Par la suite, les auteurs ont fait des évaluations de performance avec un code plus complet répondant à une problématique concrète : la conversion d'une image représentée en RVB (Rouge Vert Bleu) vers une représentation en niveaux de gris (comme l'exemple de Code 4). L'algorithme a été écrit de façon scalaire pour laisser à ICC le soin de le vectoriser automatiquement et avec Boost SIMD. Le compilateur a pu difficilement vectoriser cet algorithme plus complexe et Boost SIMD a offert, dans ce cas, de meilleures performances. Ainsi, sur SSE4.2, les auteurs de l'article ont pu observer un gain de performance entre 4,5X et 7X, tandis que sur AVX, ils ont pu mesurer un gain de performance entre 5,75X et 8,75X.

Une autre approche est d'utiliser des logiciels tels que ISPC[17] (Pharr & Mark, 2012) (*Intel SPMD Program Compiler*). Ce compilateur permet d'étendre le langage C. Il lui ajoute le support de mots clés qui permettent de décrire les vecteurs et les instructions vectorielles directement en C. Il ne

[17] https://ispc.github.io/

supporte que les jeux d'instructions vectoriels récents (d'après les années 2000 MMX et SSE ne sont donc pas supportés), ainsi que le Xeon Phi. Il s'appuie sur LLVM[18] (Lattner & Adve, 2004) pour la compilation et l'optimisation du code écrit en binaire (sauf pour le Xeon Phi). Le désavantage de cette méthode, par rapport à l'utilisation d'une bibliothèque telle que `Boost SIMD` est qu'il faut soit changer de compilateur pour pouvoir compiler le code C étendu avec ISPC, soit ajouter une étape de compilation intermédiaire pour que le code ISPC puisse être traduit en assembleur et compilé par un autre compilateur C (cette étape est nécessaire dès lors qu'on vise une compilation pour Xeon Phi). Ceci alourdit dans tous les cas le processus de compilation. Dans l'extrait de code 5, on peut voir comment faire l'exemple du code 4 avec ISPC. L'exemple est beaucoup plus proche que ce qui pourrait être fait avec un code séquentiel. La grosse différence est néanmoins l'utilisation d'un mot-clé nouveau : `foreach` qui permet de boucler sur tous les éléments.

```
foreach (uniform size_t i = 0 ... height * width)
{
    result[i] = 0.3f * red[i] + 0.59f * green[i] + 0.11f * blue[i];
}
```

Code 2-5 : Utilisation de ISPC pour faire une conversion RGB vers des niveaux de gris

Dans (Lange & Fortin, 2014) les auteurs présentent une évaluation des performances d'un algorithme de test pour la traversée d'arbres doubles, réalisée avec ISPC notamment et sur processeur Intel et Xeon Phi. Ils ont ainsi pu réaliser une étude des performances des instructions vectorielles SSE4.2 sur processeur Intel X5650, des instructions vectorielles AVX sur processeur Intel E5-2660 et du Xeon Phi 5110P en prenant pour référence une implémentation scalaire. Cet algorithme de test s'inscrit dans une application qui a pour but d'évaluer les forces d'interaction et les potentiels d'attractions entre particules. Leur première observation a été de constater qu'avec SSE et AVX, s'il n'y a pas assez de particules (moins de 6), la version scalaire offre de meilleures performances. Sur Xeon Phi, la différence se fait sentir dès que le seuil des 3 particules est franchi. En utilisant SSE, les auteurs ont réussi à obtenir des gains de performances compris entre 1,5X et 3X. En utilisant AVX, les auteurs sont parvenus à obtenir des gains de

[18] http://llvm.org/

performances entre 2,5X et 3,5X. Enfin, en utilisant le Xeon Phi, les auteurs mesurent des gains de performances entre 8X et 9X.

En 1996 quand MMX (*MultiMedia eXtensions*) est arrivé, (Peleg & Weiser, 1996) ont mesuré un gain de performances variant entre 1,5X et 2X, selon le type d'application, par rapport à l'utilisation d'un processeur sans ces instructions vectorielles. Les auteurs rapportent même des cas extrêmes où un gain de performance de 5,8X était mesuré. Plusieurs retouches ont été refaites à MMX pour le compléter, grâce à la famille des instructions SSE (*Streaming SIMD Extensions*). En 2008, un nouveau jeu d'instructions vectorielles est proposé (commercialisé début 2011) par Intel et AMD pour les processeurs à architecture x86_64 : AVX (*Advanced Vector Extensions*). Dans (Gepner, Gamayunov, & Fraser, 2011) les auteurs font une comparaison des instructions vectorielles entre SSE4.2 (architecture Nehalem – commercialisation en 2008) et AVX (architecture Sandy Bridge – commercialisation en 2011). Cette comparaison permet donc d'étudier la différence de performances entre les différentes instructions vectorielles de deux générations de processeurs qui se suivent. Elle permet aussi de simplement comparer l'efficacité de l'implémentation des mêmes jeux d'instructions de base dans ces deux générations différentes. Pour cette comparaison, les auteurs ont utilisés des processeurs de la même génération (Sandy Bridge donc) avec AVX activé ou désactivé. Dans le cas où AVX était désactivé, les exécutables retombaient sur les instructions SSE4.2. Les exécutables utilisés étaient des outils standards de *benchmark* des plates-formes de calcul à haute performance : LINPACK[19] (Dongarra, Bunch, Moler, & Stewart, 1979), STREAM[20] (McCalpin, 1995), HPCC[21] (Luszczek et al., 2006) et NPB (Bailey et al., 1991). Ils ont alors pu observer un gain de performances allant jusqu'à 1,88X, ce qui corrobore les résultats obtenus dans (Lange & Fortin, 2014). Pour les applications *memory-bound* (McCalpin, 1999), en revanche, le gain est moindre : dans tous les cas, un gain d'au moins 1,08X et au maximum 1,1X est observé.

3.1.3 – Optimisations inter-procédure

Une autre approche pour ajuster au maximum les binaires générés au microprocesseur qui devra exécuter l'application est de changer les étapes et les outils de compilation. Dans la vaste majorité des cas, un processus de compilation standard est réalisé en deux étapes : compilation intermédiaire des codes sources en un fichier binaire déjà ciblé pour la machine, puis réalisation du binaire final en agrégeant tous les fichiers binaires et en faisant l'édition des liens par un compilateur et/ou un

[19] http://www.netlib.org/linpack/
[20] http://www.cs.virginia.edu/stream/
[21] http://icl.cs.utk.edu/hpcc/

éditeur de liens. Les optimisations de code par le compilateur interviennent donc pendant la première étape de compilation et sont limitées au fichier en cours de compilation. Comme à chacune de ces optimisations le compilateur n'a connaissance que d'un seul fichier binaire, ceci interdit donc les optimisations inter-procédure (IPO : *Inter-Procedural Optimization*) (Allen, 1974). Par exemple, des fonctions non « `static` » en C ne pourront pas être fortement optimisées, étant donné que des appels pourraient être réalisés depuis d'autres fichiers. Le changement ici, consiste à demander au compilateur non plus de produire des fichiers binaires à la première étape, mais des fichiers au format Tirex (Pietrek, Bouchez, & de Dinechin, 2011). Ce format intermédiaire textuel basé sur le YAML (*YAML Ain't Markup Language*) (Ben-Kiki, Evans, & Ingerson, 2009) permet de représenter de façon portable et commune à différents compilateurs l'assembleur qui sera finalement compilé en binaire. Cela permet ensuite de modifier la seconde étape de compilation et d'utiliser un générateur de code LAO (*Linear Assembly Optimizer*) qui permettra d'optimiser le code entre les différents fichiers intermédiaires. Certains compilateurs, tels qu'ICC offrent une fonctionnalité équivalente et plus facile à mettre en œuvre pour l'utilisateur final. Il suffit d'exploiter l'option de compilation « `-ipo` » pour qu'ICC active l'IPO. Dans (Richardson & Ganapathi, 1989) les auteurs ont étudié l'impact de l'IPO, une fois qu'elle a été implémentée sur un compilateur Pascal. Ils ont pu mesurer des gains de performance entre 11% et 25% selon les applications évaluées. Ils ont également constaté que cette optimisation permet d'éviter des mouvements inutiles de données au sein des registres des processeurs étant donné que l'on acquiert la connaissance des besoins en registre des fonctions appelées. Il n'est plus nécessaire de sauvegarder les registres à l'aveugle pour conserver leur contenu au-delà d'un appel de fonction.

3.1.4 – Parallélisation

Depuis l'arrivée du SMT (*Simultaneous Multi-Threading*) dans les micro-processeurs (Tullsen, Eggers, & Levy, 1995), commercialisé sous le nom d'hyper-threading (HT) chez Intel (Marr, 2002), ceux-ci sont capables d'exécuter plusieurs threads en parallèle. Ceci a été renforcé par l'arrivée des processeurs multi-cœurs. Une application voulant exploiter au mieux un processeur doit donc tirer profit de ce niveau de parallélisme présent, d'autant plus que, aujourd'hui, les fréquences d'horloges stagnent. Dans cette section, nous nous intéresserons donc à l'exploitation de ce parallélisme d'architecture. Nous nous réfèrerons à ces techniques sous le nom de « parallélisation », que ce soit par le lancement de plusieurs processus distincts ou par le lancement plusieurs threads distincts, qu'ils communiquent ou non. Cette notion est différente de la notion classique où la parallélisation ne s'applique qu'aux simulations qui ont plusieurs processus et/ou threads (Fujimoto, 2001). Dans le

cas où il y a communication entre ces différents instances parallèles, on parle alors de distribution (Chandy & Misra, 1979). Nous avons fait ce choix de simplification dans ce chapitre, étant donné que nos propos concernent les simulations parallèles, qu'elles communiquent ou non.

Grâce aux mécanismes tels que l'ILP (*Instruction-Level Parallelism*) les processeurs et les compilateurs vont tenter de paralléliser les applications, même si elles ont été écrites de façon séquentielle. Le principe de l'ILP est de tenter, quand il n'y a pas de dépendance entre les données (comme pour la vectorisation) de paralléliser l'exécution de blocs logiques sur plusieurs cœurs de calcul. Les processeurs réalisent pour ce faire des exécutions dans le désordre (*out-of-order executions*) pour pouvoir exécuter des blocs qui se suivent mais qui sont indépendants. De la même façon, pour tenter d'accélérer l'exécution, les processeurs font de la prédiction de branches (*branch prediction*) (Smith, 1981). Quand une exécution conditionnelle doit avoir lieu dans le futur, le processeur fait une hypothèse sur la branche qui sera la plus probablement prise selon une stratégie de prédiction de branche (Smith, 1981) (Yeh & Patt, 1992) (S.-T. Pan, So, & Rahmeh, 1992) (Young & Smith, 1994) (Evers, Chang, & Patt, 1996) tentant de maximiser la validité de l'hypothèse (i.e., trouver la plus probable). Une fois l'hypothèse faite, le processeur commence à exécuter le code correspondant à cette branche (*speculative execution*). Dès que le résultat de la condition est réellement connu, si l'hypothèse était juste, alors, le programme a gagné en performances. Autrement, l'exécution en cours est abandonnée et le processeur revient à la bonne branche pour commencer l'exécution.

Des compilateurs comme ICC d'Intel fournissent également des mécanismes pour paralléliser le code automatiquement à la compilation[22], grâce à l'argument « `-parallel` » pour le compilateur Intel. Et dans ce cas, il s'agit d'une véritable parallélisation à l'aide de threads. Comme pour la vectorisation, le compilateur va se pencher sur les boucles dans le code et chercher les dépendances entre les données manipulées. Dans le cas où les données sont indépendantes, il va découper la boucle pour permettre sa parallélisation à l'exécution à l'aide d'OpenMP (Dagum & Menon, 1998).

OpenMP est justement une API (*Application Programming Interface*) qui permet à une personne de développer facilement du code parallèle. OpenMP fait appel à des `pragmas`, utilisés par le compilateur, qui permettent d'encadrer les sections du code qui doivent être parallélisées. Cela permet également d'indiquer la méthode de parallélisation, notamment comment doivent être agencés les threads à l'exécution. Le code correspondant sera généré à la compilation par le compilateur. OpenMP s'appuie sur sa propre implémentation des threads pour la parallélisation et dépend donc d'une bibliothèque partagée contenant l'implémentation pour permettre l'exécution

[22] https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers

de l'application finale. Enfin, les `pragmas` ne fournissant que des informations très limitées sur la parallélisation, pour des raisons de portabilité notamment, des variables d'environnement permettent de paramétrer l'exécution du code parallèle. Cela permet notamment d'avoir un contrôle sur le nombre de threads qui seront exécutés en parallèle. Les suites de compilateurs tels qu'Intel, GNU ou CLang supportent OpenMP pour les langages C, C++ et Fortran. Dans le cas d'Intel, le support est également assuré pour le Xeon Phi. Depuis OpenMP 4 (2013), il est également possible d'utiliser OpenMP pour développer sur GP-GPU (*General Purpose Graphical Processing Unit*). Le code 6 montre comment paralléliser simplement une boucle qui multiplie les données de deux tableaux et écrit le résultat dans un autre tableau.

```
#pragma omp parallel for
for (unsigned int i = 0; i < maxSize; ++i)
{
    tab3[i] = tab2[i] * tab1[i];
}
```

Code 2-6 : Utilisation d'OpenMP pour paralléliser un traitement de tableau

Ainsi, dans (Delisle, Krajecki, Gravel, & Gagné, 2001), les auteurs évaluent les performances d'un algorithme parallélisé avec OpenMP. Il s'agit d'une méta-heuristique d'optimisation utilisant un algorithme de colonies de fourmis, qu'ils parallélisent. Dans leur article, les auteurs observent, selon les conditions dans lesquelles l'algorithme est lancé, un gain de performance entre 5X et 6X par rapport à l'implémentation séquentielle. Néanmoins, ce gain est constaté sur 8 cœurs d'exécution et rajouter des processeurs ne permet pas de gagner en performances. Les auteurs calculent également l'efficacité des processus et constatent notamment que cette efficacité décroît en fonction du nombre de processeurs et l'efficacité devient particulièrement mauvaise une fois que le pic des 8 cœurs est atteint. L'efficacité n'est pas non plus maximale avec 8 cœurs, malgré le gain de performance maximal. En effet, il aurait fallu avoir un gain de performance de 8X pour avoir une efficacité de 8. Ces problématiques seront discutées plus en détails dans la section 4. Dans (Dedu, Vialle, & Timsit, 2000), les auteurs ont montré que pour les algorithmes simples, OpenMP fournit de

meilleures performances autant en terme de temps de développement qu'en terme de temps de calcul que la bibliothèque pthread. Cependant, pour les algorithmes plus complexes à paralléliser, alors ils conseillent d'utiliser pthread.

Néanmoins, dans (Mazouz, Touati, & Barthou, 2010) (Mazouz, Touati, & Barthou, 2011), les auteurs ont montré que les performances d'OpenMP étaient extrêmement variables. Ils ont montré que si l'exécution de leur application test (SPEC CPU 2006) était stable dans le temps quand elle est exécutée sur CPU de façon séquentielle (une variation temporelle de l'ordre de 1% du temps d'exécution était observée), il n'en était pas de même avec OpenMP sur une machine multi-cœur. Dans le cas d'OpenMP multi cœurs, selon les applications de test (SPEC OMP 2001), les auteurs observent des temps d'exécution qui peuvent prendre jusqu'à 50% de temps supplémentaire d'un lancement à l'autre et ce quel que soit le compilateur employé (GCC ou ICC). Ces observations démontrent la difficulté à faire des évaluations correctes des performances d'applications parallélisées avec OpenMP, ainsi que l'absence totale de signification d'une durée moyenne d'exécution d'une application OpenMP une fois exécutée en parallèle. L'enseignement tiré de cette étude est notamment l'importance de lancer plusieurs réplications de la même application afin d'en évaluer les performances et de ne pas s'arrêter sur un résultat unique, qui aurait peu de signification. Cela est d'autant plus critique que les applications SPEC sont écrites pour être hautement répétables et stables. Dans (Mazouz et al., 2011), les auteurs apportent une explication au phénomène observé : OpenMP est extrêmement sensible à sa « *thread affinity* ». La « *thread affinity* » dans OpenMP permet de définir selon quelle politique (dans quel ordre) les threads vont être affectés sur les différents cœurs logiques disponibles sur la machine. Lors de la fabrication d'un processeur, celui-ci a un nombre défini (et fixe) de cœurs physiques, gravés dans le silicium. Par la suite, grâce à une implémentation logicielle dans le processeur, il est possible de dédoubler ou quadrupler le nombre de threads que peut lancer un cœur physique, en occupant les espaces libres dans le pipeline du cœur. Ces threads « intercalés » sont alors considérés comme tournant sur des cœurs logiques. Les processeurs modernes ont généralement un

potentiel de deux threads par cœur physique ; c'est-à-dire un cœur logique et un cœur physique. Par défaut, aucune affinité n'est définie et les threads se promènent sur les cœurs selon les mouvements définis par l'ordonnanceur du système d'exploitation. Deux affinités sont disponibles, la compacte (« *compact* ») et l'éparpillée (« *scatter* »). Dans les deux cas, les threads sont alors affectés de manière fixe à un cœur logique et ne peuvent plus en bouger. Les affinités changent juste la politique d'allocation. Dans le cas de la « compact », les threads sont affectés par CPUs entiers. OpenMP rempli d'abord tous les cœurs logiques d'un CPU avant de remplir le CPU suivant. Dans le cas de la « *scatter* », OpenMP affecte un thread par cœur logique par CPU, puis une fois que tous les CPUs sont « chargés » avec un thread, il réaffecte le thread suivant sur le cœur logique suivant du 1er CPU et ainsi de suite. Dans le cas où une affinité est définie, qu'elle soit « *scatter* » ou « *compact* », les auteurs ont ainsi pu observer que les fluctuations cessaient et le temps de calcul devenait stable et reproductible. Ils ont également observé qu'avec la politique « *scatter* » ils obtenaient de meilleures performances : étant donné que les processeurs n'étaient pas surchargés, les caches de niveau 2 étaient mieux exploités. Néanmoins, définir une affinité n'a pas stabilisé toutes les applications de test de SPEC OMP 2001. Les auteurs ont ainsi pu déterminer que lancer des processus annexes, non relatifs à OpenMP, stabilisait également les temps d'exécution, en réduisant la compétition entre les threads OpenMP pour accéder à leurs données en cache.

De la même façon qu'OpenMP, d'autres standards pour faciliter la parallélisation de code ont été proposés : OpenACC (*Open ACCelerators*) (OpenACC Working Group, 2011) d'une part et OpenHMPP (*Open Hybrid Multicore Parallel Programming*) (OpenHMPP Consortium, 2011) d'autre part. Les deux s'appuient sur des `pragmas` de compilateurs pour permettre la parallélisation du code. OpenACC est développé par Cray, CAPS (*Compiler and Architecture for Embedded and Superscalar Processors*), PGI (*The Portland Group Inc*) et NVIDIA. Cray, CAPS et PGI proposent un compilateur commercial pour développer avec OpenACC. GCC 5 supporte également OpenACC. La différence majeure entre OpenACC et OpenMP est qu'OpenACC supporte les architectures hétérogènes : CPU et GPGPU. Et c'est ce qui a motivé sa création, puisqu'à l'époque OpenMP ne supportait pas encore les GPGPU. De

même que pour OpenMP, OpenACC supporte le C, le C++ et le Fortran. Le code 7 montre la même parallélisation de boucle que pour le code 6, mais avec OpenACC.

```
#pragma acc parallel loop
for (unsigned int i = 0; i < maxSize; ++i)
{
    tab3[i] = tab2[i] * tab1[i];
}
```

Code 2-7 : Utilisation d'OpenACC pour paralléliser un traitement de tableau

Cependant, dans (Liao, Yan, de Supinski, Quinlan, & Chapman, 2013), en voulant évaluer les performances de leur implémentation d'OpenMP4 s'appuyant sur CUDA (*Compute Unified Device Architecture*) (Garland et al., 2008), les auteurs de l'article constatent d'importantes disparités de performances entre les deux compilateurs OpenACC évalués. En effet, pour évaluer les performances de leur implémentation, ils évaluent les performances avec le compilateur PGI OpenACC et le compilateur HMPP OpenACC. Le compilateur HMPP OpenACC semble toujours offrir de meilleures performances que le compilateur PGI OpenACC avec des gains de performance compris entre 1.5X et 20X !

Dans (Xu, Chandrasekaran, & Chapman, 2013) les auteurs proposent une autre approche pour l'utilisation d'OpenACC. En effet, ils proposent d'utiliser conjointement OpenACC et OpenMP pour pouvoir faire du calcul hybride exploitant au maximum les CPUs et les GPUs disponibles sur une machine. Pour ce faire, dans leur code, ils parallélisent des sections sur CPU, à l'aide des directives OpenMP, puis, dans ces portions de code parallèle, ils réalisent d'autres boucles parallèles à l'aide d'OpenACC pour pouvoir distribuer leur code sur un ou deux GPUs. S'ils n'ont pas fourni d'évaluation des performances en comparaison avec une implémentation séquentielle de leurs applications de tests, ils ont fourni des comparaisons avec un ou deux GPUs exploités. Selon la taille des données traitées durant les tests, ils obtiennent, lors du passage à deux GPUs, un gain de performances entre 1X (petit jeu de données) et 1,90X. Pour l'une des applications de test (dans le domaine de la thermodynamique),

les auteurs parviennent à obtenir un gain de performance de 2,2X en rajoutant un GPU : c'est-à-dire qu'ils se retrouvent en présence d'une application super linéaire dans ce contexte.

De son côté, OpenHMPP est développé par CAPS. Son objectif est de permettre d'écrire simplement du code pour des accélérateurs matériels. A l'heure actuelle, seuls les GP-GPUs de NVIDIA sont supportés. Il n'est donc pas question de CPU ici. Il s'appuie également sur des pragmas pour permettre de conceptualiser le parallélisme du code : transfert de données, noyau d'exécution parallèle, etc. Deux compilateurs commerciaux sont disponibles, un distribué par CAPS et l'autre par ENZO. Dans les deux cas, pour permettre d'exploiter le maximum de performances des GP-GPUs, les compilateurs s'appuient sur l'infrastructure CUDA de NVIDIA (Garland et al., 2008). Cet acronyme désigne une architecture et un langage qui permettent de développer des applications pour les GP-GPUs qui le supportent. Pour ce qui est du langage, CUDA propose un dérivé du langage C++ qui permet *via* de nouveaux mots clés et de nouvelles fonctions, d'écrire des « *kernels* » : ce sont des fonctions qui constituent des noyaux d'exécutions parallèles qui seront exécutés sur le GP-GPU. *Via* ce langage, il est également possible de gérer les transferts de données entre l'hôte et l'accélérateur matériel. Cette technologie, propriétaire, ne permet pas de s'interfacer avec les accélérateurs matériels autres que ceux de NVIDIA. Les GP-GPUs d'AMD sont donc exclus et l'architecture MIC d'Intel également.

Dans (Falcou, Sérot, Chateau, & Lapresté, 2006), les auteurs proposent également une autre approche pour paralléliser le code grâce à l'utilisation des *templates* du langage C++. C'est une approche similaire à celle que les auteurs ont également développé pour `Boost SIMD`, tout étant réalisé à la compilation par le compilateur. C'est cette même approche qu'on retrouve dans Intel TBB (*Threading Building Blocks*) (Pheatt, 2008), proposée avec les compilateurs Intel. Cette bibliothèque permet également de paralléliser du code grâce aux *templates* du langage C++. On retrouve une approche similaire dans la bibliothèque C++ MTPS (Kirschenmann, Plagne, & Vialle, 2012) qui s'appuie notamment sur ses propres types, sa propre représentation des données pour pouvoir paralléliser efficacement le code, que ce soit sur le matériel (SIMD) ou par le logiciel (OpenMP).

3.1.5 – Distribution

Une fois l'application parallélisée sur un nœud local, par quelque méthode que ce soit, il est également possible de la distribuer sur plusieurs nœuds afin d'augmenter les performances de

l'application. Ici, le terme distribution désigne donc le simple fait d'avoir une application parallèle (mono/multi processus, *mono*/*multi-threads*) exécutée sur plusieurs nœuds de calcul en parallèle, qu'il y ait communication entre les nœuds ou non. Comme expliqué précédemment, nous sortons de la définition classique. Dans la définition classique (Chandy & Misra, 1979), le simple fait qu'il y ait communication entre les instances signifie que l'on a une simulation distribuée. Cependant, étant donné que dans l'intégralité de notre thèse, nous ne traiterons que de simulations parallèles sans communication, nous avons fait le choix de nous réapproprier les termes, afin de pouvoir désigner de façon plus précise les travaux que nous réalisons : travail sur un unique nœud de calcul (parallèlisation), sur plusieurs nœuds de calcul (distribution). Ceci nous permet de simplifier la distinction parallélisation « locale » et parallélisation « sur plusieurs nœuds ». Dans certains cas, la distribution est très simple, il suffit de lancer une instance par nœud puis d'agréger les résultats. Dans d'autres cas, il est nécessaire que l'intégralité de l'application puisse communiquer sur chacun des nœuds. Dans ce cas, des outils tels que MPI (*Message Passing Interface*) (Snir et al., 1995) permettent de répondre à la problématique. Cette API permet d'échanger des messages et de gérer la distribution des instances de l'application à travers un cluster. Cependant, pour avoir une bonne utilisation de MPI qui ne grève pas les performances de l'application, il est nécessaire de définir un bon plan du réseau de communication : comment les instances vont-elles communiquer entre elles ? L'usage de MPI ne proscrit pas l'utilisation d'autres API telles qu'OpenMP pour la parallélisation de l'application sur les nœuds. On retrouve donc fréquemment le duo MPI/OpenMP dans les applications scientifiques. Ces deux APIs sont supportées sur le Xeon Phi.

3.2 – Du bon usage de la mémoire

Si le traitement des données est extrêmement important pour avoir une application performante et optimisée, il faut que les données soient bien agencées et accessibles pour permettre à cette performance de s'exprimer. Il est donc nécessaire de réfléchir à la façon dont seront stockées les données en mémoire. Ainsi, pour pouvoir faciliter la vectorisation, il faut au maximum éviter les tableaux de structures, pour utiliser à la place des structures de tableaux. Dans l'écriture du code, cela change peu de choses même si c'est une écriture moins intuitive, moins naturelle. En effet, on retrouve les mêmes éléments, seul l'ordre change. Pour le compilateur, en revanche, cela lui permettra de manipuler des tableaux de valeurs et donc, potentiellement de pouvoir vectoriser automatiquement le code. On peut retrouver un exemple sur les Code 8 et Code 9. Dans le premier cas, on y défini un tableau de structures, que le compilateur ne pourra pas optimiser. Dans le second

cas, les données sont représentées avec une structure de tableau. Dans les deux cas sont toujours stockés les mêmes éléments : la charge et la position dans l'espace de N particules.

```
struct particule {
    float charge ;
    float x ;
    float y ;
    float z ;
} ;
particules[N];
```

Code 2-8 : Tableau de structures pour représenter N particules et leur charge

```
struct particules {
    int N ;
    float * charges ;
    float * x ;
    float * y ;
    float * z ;
} ;
```

Code 2-9 : Structure de tableaux pour représenter N particules et leur charge

Par ailleurs, dans le cadre de la vectorisation, les processeurs sont plus efficaces si les adresses des données sont alignées, c'est-à-dire multiples d'une certaine valeur. Dans le cadre du Xeon Phi, il s'agit de 64 bits. Le développeur a donc la responsabilité de vérifier que ses données sont correctement alignées, en ajoutant par exemple du remplissage (« *padding* ») en cas de besoin.

De plus, un nœud de calcul dispose de plusieurs types de mémoires, dont les tailles et les vitesses d'accès varient. Plus on s'éloigne du processeur, plus les mémoires deviennent lentes mais plus leur capacité augmente. Ainsi, la mémoire la plus rapide mais la plus petite en taille est la mémoire des registres du CPU. Viennent ensuite les caches du processeur, de niveau 1 (L1), niveau 2 (L2) puis éventuellement de niveau 3 (L3). Enfin, on accède aux mémoires hors CPU, donc, beaucoup plus lentes, mais bien plus volumineuses telles que la RAM, les disques durs puis les disques réseau. Pour les performances de l'application, il est donc critique d'exploiter au mieux les mémoires les plus rapides. Il convient donc d'optimiser au maximum les applications pour que les données restent dans les mémoires rapides pendant tout le temps où elles sont nécessaires, pour éviter d'avoir à aller les rechercher. On travaille donc au maximum avec les caches du processeur, la condition à

satisfaire étant que, dès qu'une application a besoin d'une donnée, celle-ci soit déjà présente en cache. Dans ce cas, on parle de « *cache hit* ». Si en revanche, la donnée n'est pas présente en cache, on parle alors de « *cache miss* » (défaut de cache). Les défauts de cache peuvent être classés en trois catégories. Il y a les « *compulsory misses* » (ou « *cold misses* ») qui correspondent à la première tentative d'un programme d'accéder à une zone mémoire. Celle-ci n'ayant jamais été accédée précédemment, un défaut de cache a lieu. Il est possible de réduire ces défauts de cache, grâce au « *prefetching* » (Berg, 2002) qui permet d'amener les données en cache avant qu'elles ne soient nécessaires. Les défauts de cache de la deuxième catégorie sont les « *capacity misses* ». Ces défauts de caches apparaissent quand des données nécessaires ne peuvent tenir en cache. Enfin, on trouve les « *conflict misses* ». Ces défauts de cache sont ceux sur lesquels on peut travailler pour l'optimisation. Ils apparaissent quand la donnée à laquelle on veut accéder a été expulsée du cache par une autre donnée concurrente. Il est donc important d'éviter que des données ne rentrent en conflit pour le cache. Dès lors qu'un défaut de cache se produit, il y a une perte de performance, puisqu'il faut aller récupérer la donnée sur une mémoire plus lente (RAM, disque, etc). Quand ces « *caches misses* » deviennent trop important et empêchent l'application d'exploiter au maximum les possibilités de calcul de la machine puisqu'elle perd la majorité de son temps à attendre ses données, on parle alors d'application « *memory-bound* » (McCalpin, 1999); son facteur limitant est l'accès aux données, et non le calcul sur lesdites données. On parle également de « *memory wall* » (Wulf & McKee, 1995); les performances de calcul des processeurs ayant progressé bien plus vite que les performances d'accès à la mémoire.

On trouve un exemple d'optimisation de l'utilisation des caches dans (Frigo, Leiserson, Prokop, & Ramachandran, 1999) avec la multiplication de matrices. Dans un algorithme naïf, le produit matriciel consiste, pour chaque ligne de la première matrice et chaque colonne de la seconde matrice, à multiplier terme à terme leurs éléments puis à faire la somme des résultats de ces multiplications. Ainsi pour calculer le produit de deux matrices carrées 4x4, on se retrouve à faire 48 sommes et 64 multiplications pour obtenir la matrice résultat 4x4. Si on implémente le produit matriciel de cette façon en informatique, on constatera qu'il y a une très mauvaise utilisation des caches. Dans la majorité des cas, une matrice est représentée en mémoire ligne par ligne. Si on garde une ligne de la première matrice en cache sur plusieurs itérations, en bouclant sur les colonnes de la seconde matrice, on se retrouve obligé d'accéder aux colonnes de la seconde matrice, et donc à changer de ligne dans celle-ci à chaque itération. On peut aussi retrouver des cas où la représentation est colonne par colonne, mais on se heurte tout de même au même problème : pour l'une des deux matrices, on accédera aux éléments de la matrice dans un ordre qui va à l'encontre de sa représentation en mémoire. Ceci induit une perte de performances en raison de tous les

« *cache misses* » qui ont lieu. La proposition de (Frigo et al., 1999) est de transposer une des deux matrices (la transposée d'une matrice étant obtenue en intervertissant lignes et colonnes). Dans ce cas, pour le calcul, on se retrouve à travailler uniquement sur des lignes de matrices (ou des colonnes, selon la représentation).

Dans la majorité des cas, les compilateurs tenteront d'optimiser au maximum le code produit pour exploiter les caches de façon optimisée. De nombreux travaux ont été réalisés dans ce sens et des motifs et des techniques d'optimisations ont été mis au point : modification des boucles (McKinley, Carr, & Tseng, 1996) ; « *array padding* » (Rivera & Tseng, 1998) ; « *loop tiling* » (Ghosh, Martonosi, & Malik, 2000) ; « *loop fission* » (Wang, Sha, & Hu, 2001) ; « *optimization orchestration* » (Z. Pan & Eigenmann, 2006) ; « *code features prediction* » (Dubach et al., 2007). Ces techniques ont été développées pour améliorer la prise en charge des caches dans les compilateurs, soit en manipulant directement la représentation des données en mémoire, soit en modifiant les heuristiques suivies par le compilateur pour générer le code machine manipulant les données.

Dans (Tao, 2010) l'auteur présente des résultats d'optimisation d'une application après son profilage. Les optimisations n'ont porté que sur l'utilisation des caches, en exploitant certaines des méthodes citées plus haut. L'auteur a effectué ses tests sur plusieurs architectures (Intel Pentium 3 et 4, Intel Itanium, ainsi qu'un processeur AMD) et sur des jeux de données de tailles différentes. Lors de l'utilisation du « *prefetching* » (Berg, 2002), l'auteur est parvenu à obtenir un gain de performance de 1X à 1,8X. L'utilisation du « *array padding* » (Rivera & Tseng, 1998), de la « *loop fission* » (Wang et al., 2001) ont amené des gains de performance respectifs de 1X à 3,2X et de 0,9X et 2,4X. De ce test, il apparaît également que l'optimisation *via* cette méthode est extrêmement sensible à la taille des données ainsi qu'au processeur. Le processeur AMD est celui qui présente les meilleurs gains de performance. Enfin, en exploitant le « *loop tiling* » (Ghosh et al., 2000), l'auteur a obtenu des gains de performance de 0,8X à 1,3X. Il apparaît clairement dans cette étude de l'optimisation de l'utilisation des caches que le mieux peut être l'ennemi du bien. Si de bons gains de performance peuvent être obtenus, on constate que tenter d'optimiser la gestion des caches peut parfois conduire à une baisse de performances, ce qui nous ramène à (Kernighan & Plauger, 1978).

3.3 – Du bon usage du matériel

Une fois l'application optimisée sur le CPU cible, il peut être nécessaire de changer d'architecture matérielle pour tenter d'obtenir des performances encore meilleures. Il y a peu, les seuls architectures utilisées en dehors des CPUs étaient les GP-GPUs (*Generic Purpose Graphical*

Processing Unit) dont le constructeur principal est Nvidia. Ces accélérateurs matériels, qui étaient à l'origine des cartes graphiques, ont une importante capacité de calcul parallèle et vectorielle. Différentes méthodes, présentées dans les sections précédentes permettent d'écrire un code qui est à la fois exécutable sur CPU ou sur GP-GPU. Pour les cartes d'origines Nvidia, même si il est préférable d'utiliser leur propre technologie, à savoir CUDA (Garland et al., 2008), il existe une alternative plus portable nommée OpenCL (Stone, Gohara, & Shi, 2010). OpenCL s'appuie sur une notion d'abstraction du matériel. Il faut écrire un code parallèle qui pourra s'exécuter sur différentes architectures dans un *kernel* OpenCL. Le *kernel* doit être compilé avant son exécution par un pilote dédié au matériel cible. Ce dernier pouvant être un simple CPU, un GP-GPU, ou encore un Xeon Phi. A partir du moment où un pilote existe pour compiler le code OpenCL en code natif, l'architecture matérielle est supportée par OpenCL. Les performances sont limitées par la généricité du code écrit, ainsi que par la qualité du pilote OpenCL disponible pour l'architecture. A titre d'exemple, dans (Venetis, Goumas, Geveler, & Ribbrock, 2014), les auteurs ont pu constater les maigres performances du pilote OpenCL pour Xeon Phi, leur solution pour obtenir de réelles performances avec le Xeon Phi ayant été de se passer d'OpenCL pour utiliser OpenMP.

Cependant, comme montré dans les sections précédentes, les outils pour travailler sur Xeon Phi sont de plus en plus nombreux et ne se limitent pas à la simple utilisation d'OpenCL, notamment en ce concerne les outils de profilage.

Néanmoins, les performances du Xeon Phi ne sont pas encore tout à fait au rendez-vous, comparé à cellles des GP-GPUs de NVIDIA. Dans (Bernaschi, Bisson, & Salvadore, 2014) les auteurs font l'évaluation des performances d'une simulation de physique sur trois architectures différentes et distribuées : des processeurs Sandy Bridge (Intel E5-2658 pour les machines avec les Xeon Phi, Intel E5-2687W pour les machines avec les GP-GPUs), des cartes GPGPU NVIDIA Kepler K20s (architecture Kepler) ainsi que des Xeon Phi 5110P. Les auteurs expliquent que la simulation a été hautement optimisée pour GPU et pour processeur d'abord, et ensuite pour Xeon Phi. Ce dernier effort a été nécessaire pour exploiter au mieux les nombreux threads disponibles sur la plate-forme, ainsi que pour exploiter correctement les instructions vectorielles. Par ailleurs, le Xeon Phi n'ayant pas d'espace de stockage embarqué, mais seulement un système de fichiers réseau (NFS), il faut être très prudent avec les performances des applications utilisant des volumes de données significatifs, en tentant de réduire au maximum les entrées / sorties. Dans un premier temps, les auteurs ont constaté la sensibilité du Xeon Phi au « *padding* » des données. En raison de la façon dont est implémenté le cache de niveau 2 pour le TLB (*Translation Lookaside Buffer*) (Stravers & Van De, 2004), il leur était nécessaire d'ajouter beaucoup de données vides dans leur tableaux de données pour aligner correctement la mémoire. Ils ont pu constater qu'un mauvais alignement pouvait

causer une perte de performance jusqu'à 0,5X. Par ailleurs, les auteurs ont pu constater que le Xeon Phi offre un facteur de passage à l'échelle plus limité, comparativement aux GPGPUs de la génération Kepler. Enfin, les auteurs concluent que sur leur architecture d'étude, avec le code le plus optimisé dont ils disposaient, un GP-GPU ou un Xeon Phi offrent les performances de cinq de leurs processeurs. En revanche, quand on se place dans un contexte de multi-GP-GPU ou de multi-Xeon Phi, les auteurs concluent que si l'application permet un passage à l'échelle important, un GP-GPU de type Kepler K20 vaut entre 1,5 et 3,3 Xeon Phi 5110P, tandis que si l'application ne permet pas un passage à l'échelle important, alors un GP-GPU K20 vaut environ 2 Xeon Phi 5110 P.

En revanche, dans (Lyakh, 2015) l'auteur donne un avis plus nuancé sur les performances du GP-GPU et du Xeon Phi. En effet, l'auteur constate de fortes disparités dans les performances des deux accélérateurs. Dans son étude, il utilise une carte Tesla K20X (architecture Kepler) ainsi qu'un Xeon Phi 5110P. Les deux offrent de meilleures performances que les différents processeurs qu'il teste, l'AMD 6378 et l'Intel Xeon E5-2670. En revanche, en termes de performances, il est difficile de départager les deux architectures. Si les performances ne sont pas égales, elles dépendent fortement de ce qui est demandé, des contraintes algorithmiques et des jeux de données considérés. L'auteur note néanmoins qu'il était plus facile d'optimiser l'utilisation des caches de niveau 1 sur un processeur classique (et donc sur un Xeon Phi) que de réaliser des accès mémoire coalescents sur le GP-GPU, alors que ceux-ci sont critiques pour les performances des applications sur GP-GPU. Ceci a néanmoins permis d'atteindre un gain de performance entre 2X et 3X sur GP-GPU.

Dans (Murano et al., 2014) les auteurs comparent à nouveau les performances sur un même algorithme entre une carte GP-GPU NVIDIA K20, un Xeon Phi 5110P et deux processeurs Intel Xeon E5-2643. Ils ont ainsi pu obtenir un gain de performance entre 8X et 16X simplement entre les processeurs et le GP-GPU. En revanche, les gains de performance entre le Xeon Phi et les processeurs étaient moindres : ils étaient entre 2,4X et 13X. Néanmoins, ces résultats sont à mettre en perspective par rapport aux deux articles précédents dans lesquels les différences entre Xeon Phi et GP-GPU étaient plus importantes que dans celui-ci. Ici, les auteurs reconnaissent qu'aucun effort particulier n'a été fait pour porter l'application sur Xeon Phi, contrairement à ce qui avait été réalisé dans les deux autres articles. Ils ont juste utilisé les pragmas de compilation pour utiliser le Xeon Phi en « *offload mode* ». Ce qui signifie qu'ils sont probablement tombés sur les problèmes mis en lumière dans les articles précédents : il est nécessaire de vectoriser correctement et de gérer correctement sa mémoire sur Xeon Phi, faute de quoi les performances tombent.

Dans (Crimi, Mantovani, Pivanti, Schifano, & Tripiccione, 2013) les auteurs comparent les performances brutes de deux algorithmes sur un Xeon Phi de pré-production (KNC – 61 cœurs

cadencés à 1,09 GHz), sur un GP-GPU NVIDIA Tesla C2050 (architecture Fermi précédant l'architecture Kepler) ainsi que sur deux dual-processeurs Westmere et Sandy Bridge dont les modèles précis ne sont pas donnés. Les deux algorithmes présentent deux caractéristiques différentes. Tandis que l'un est plus porté sur la manipulation de la mémoire, le second est purement calculatoire. Ils ont ainsi obtenu des résultats totalement différents. Pour le premier algorithme, le GP-GPU présente les meilleures performances. Comparé aux CPU les plus lents (les Westmere), il présente un gain de performance de 4,8X. Le Xeon Phi, lui, ne vient qu'en troisième position avec un gain de performance de 3X, battu par le dual-Sandy Bridge avec un gain de performance de 3,4X. Par ailleurs, les auteurs ont également fait l'évaluation de la performance relative de l'architecture comparativement à sa performance maximale théorique qu'ils désignent sous le terme d'efficacité (« *efficiency* »). Ils constatent ainsi que le Xeon Phi n'est exploité qu'à 16% de sa puissance, tandis que le dual-Sandy Bridge l'est à 70%, et le GP-GPU à 58%. En revanche, pour le second algorithme, purement calculatoire, le Xeon Phi reprend la tête avec un gain de performance de 3,1X par rapport aux processeurs les plus lents (les Westmere), et avec une efficacité de 27%. Ensuite, arrivent le dual-Sandy Bridge avec un gain de performance de 2,5X et une efficacité de 63%, et enfin le GP-GPU avec un gain de performance de 2,3X et une efficacité de 41%. De leur étude, il ressort qu'exploiter correctement un accélérateur matériel n'est pas une évidence. Aucun accélérateur (GP-GPU ou Xeon Phi) n'atteint l'efficacité de 70% qui a été obtenue avec les deux Sandy Bridge. Par ailleurs, à la vue des premières performances obtenues, la mémoire semble également être moins efficace sur Xeon Phi que sur GP-GPU. Cependant, les performances décrites dans cette étude sont à nuancer pour plusieurs raisons : elles datent de 2013 et le matériel utilisé est quelque peu obsolète pour le monde du calcul hautes-performances ; l'architecture Fermi des GP-GPUs a été remplacée par l'architecture Kepler ; de même, le Xeon Phi utilisé est un Xeon Phi de pré-production, un peu atypique, dont les caractéristiques ne correspondent pas à celles des modèles commercialisés aujourd'hui.

Dans (Venetis et al., 2014), les auteurs ont comparés l'implémentation de deux *kernels* de calcul, l'un manipulant beaucoup de données, l'autre moins. Ils ont comparé l'exécution de la même implémentation, s'appuyant sur OpenMP. Pour cette comparaison, ils ont utilisé deux architectures cibles : un dual processeur Intel Sandy-Bridge E5-2658 à 2,10 GHz (8 cœurs chacun) ainsi qu'un dual Intel Xeon Phi 5120D à 1,053 GHz (60 cœurs). Le *kernel* purement calculatoire a été le plus efficace sur Xeon Phi, fournissant un gain de performances de 4,68X. Pour le *kernel* manipulant beaucoup de données (des matrices), les performances n'étaient pas égales et dépendaient du jeu de données : pour l'un, le gain de performance était de 1,15X en faveur des Xeon Phi, tandis que pour l'autre, il était de 1,20X en faveur des Sandy-Bridge.

On peut ainsi constater, dans ces différentes études que les ambitions d'Intel ne sont pas tout à fait atteintes à l'heure actuelle. Le Xeon Phi ne présente pas d'aussi bonnes performances que prévu, surtout comparé à un GP-GPU récent de NVIDIA. D'autant plus qu'un minimum de travail est nécessaire en amont pour porter correctement l'application sur Xeon Phi, faute de quoi le Xeon Phi offre des performances moindres que celles que l'on peut obtenir avec certains CPUs. Ceci minimise grandement l'argument selon lequel le Xeon Phi compense sa plus faible puissance par un temps de portage réduit. Néanmoins, dans la comparaison, il faut prendre en compte le temps de portage réellement important qui est nécessaire pour les GP-GPUs, ainsi que le besoin de connaître correctement cette dernière architecture matérielle pour pouvoir l'exploiter au mieux. D'autant plus que pour les applications non vectorisables, le GP-GPU n'apporte quasiment aucun gain de performance, alors que le Xeon Phi peut apporter plus de cœurs.

Néanmoins, on constate dans (Dokulil et al., 2013) que, dans le cadre du calcul hydride, où l'on exploite à la fois les Xeon Phi et les processeurs présents sur un nœud de calcul, on peut atteindre de très bons gains de performances. Ainsi, pour une simulation en physique des particules, les auteurs de cet article ont pu, en exploitant les deux Xeon Phi (modèle de pré-production, comme décrit précédemment) et tous les processeurs Intel Xeon X5680 de leur nœud de calcul, obtenir un gain de performance de 27,3X par rapport à une exécution séquentielle. De même, le simple fait d'utiliser tous les cœurs disponibles et un Xeon Phi leur a permis d'atteindre un gain de performance de plus de 2X, et de plus de 2,75X quand ils ont exploité tous les cœurs et les deux Xeon Phis. Dans le cas où seul un des Xeon Phi était utilisé, celui-ci avait la responsabilité de 54% des particules, tandis que dans le cas où les deux étaient utilisés, ils prenaient 71% de la charge de travail. L'avantage des accélérateurs matériels apparaît clairement ici. Néanmoins, ces résultats, s'ils sont très bons en termes de temps de calcul, sont à mettre en perspective avec l'absence d'analyse vis-à-vis de l'exactitude des résultats obtenus. Sont-ils les mêmes que ceux qui avaient déjà été obtenus sur un processeur classique ? Cette problématique, bien réelle, nous amène aux limites de l'optimisation et à ses dangers potentiels.

4 – Limites de l'optimisation

Même si de prime abord l'optimisation semble bénéfique, elle peut amener de nouvelles problématiques, voire ne pas être applicable en pratique. Comme constaté dans certaines des publications précédentes, tenter d'optimiser, et donc, forcer le compilateur à des comportements spécifiques le conduit parfois à produire un code moins optimisé que ce qu'il produisait déjà.

Les limites de l'optimisation peuvent surtout s'observer et se faire sentir dans le cas où l'on parallélise une application, en raison notamment des sections de code qui ne sont pas facilement parallélisables ou qui sont liées à des problématiques d'exclusion mutuelle : dans ce dernier cas, le temps gagné à calculer en parallèle est perdu à attendre la disponibilité de données. Enfin, une dernière limite observable est que à trop forcer les optimisations, on peut se retrouver à calculer bien plus vite… quelque chose qui n'a plus rien à voir avec le résultat que l'on doit obtenir ! Nous reviendrons sur ces aspects lorsque nous parlerons de reproductibilité numérique.

4.1 – Portions séquentielles, interprocessus

Dans le cas des applications parallélisées ou distribuées, il est possible que des sections ne puissent pas être parallélisées et représentent donc des goulets d'étranglement séquentiels, ou qu'elles soient exécutées dans chacune des instances distribuées et représentent donc un travail identique dupliqué. Ce sont des portions de code purement séquentielles qui sont exécutées de manière inconditionnelle. Ces portions de code limitent donc le gain de performance maximal qu'on pourrait attendre de l'application. Ces contraintes sont définies par la loi d'Amdahl (Amdahl, 1967) qui exprime le gain de performance maximum que l'on peut attendre d'une application connaissant son pourcentage de code séquentiel et le nombre de cœurs sur laquelle on va la paralléliser/distribuer, à quantité de données constantes. La loi d'Amdahl met en évidence des paliers dans le gain de performance apporté par la parallélisation. *De facto*, à partir d'un certain point, qui dépend de la quantité de code séquentiel, l'ajout de cœurs n'aura plus aucun impact sur les performances et aucun gain supplémentaire ne sera observé, d'où la notion de gain de performance maximum qui est simplement l'ordonnée maximale atteinte par la courbe des performances lorsqu'elle atteint son palier. Ces paliers peuvent être observés sur la Figure 7 qui reproduit les gains de performance maximum qui peuvent être obtenus selon la quantité de code séquentiel et le nombre de cœurs sur lesquels l'application s'exécute. Ces paliers peuvent s'expliquer notamment par le fait qu'une machine n'est pas idéalement parallèle. Si par exemple, elle possède deux cœurs, le parallélisme ne sera pas parfait pour des raisons matérielles (bus, dépendances, mémoire, etc) et par conséquent dans la majorité des cas, l'application n'aura pas un gain de performance de 2X.

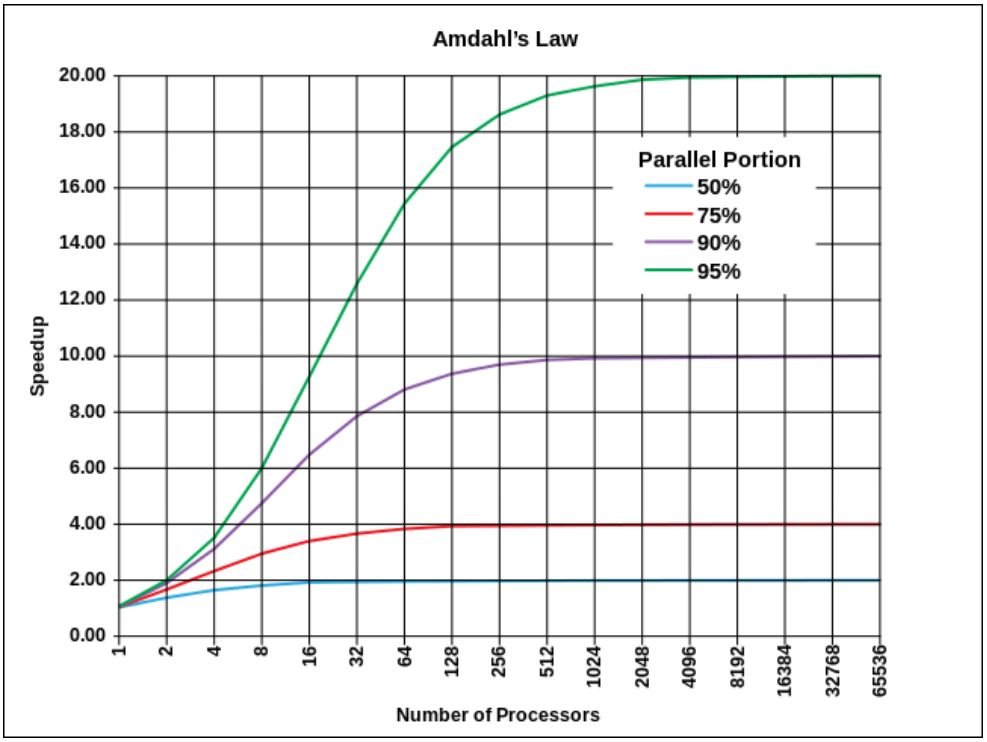

Figure 2-7 Gain de performance maximum qui peuvent être atteint selon la loi d'Amdahl (http://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg)

Par ailleurs, dans le cadre d'une application distribuée ou parallèle, des instances ou des processus peuvent avoir besoin de communiquer entre eux, qu'ils soient sur le même hôte ou sur deux machines différentes. Dans ce cas, des outils tels que MPI permettent de faciliter le travail. Néanmoins ces communications ont un coût d'autant plus élevée qu'elles sont distantes et qu'elles utilisent de réseau. Ainsi, dans (Brandfass, Alrutz, & Gerhold, 2013) les auteurs mesurent les performances brutes des communications MPI sur un réseau Infiniband *Single Data Rate* (SDR) qui offre une bande passante de 1 280 Mo/s (Pfister, 2001), sans optimisation. Ils observent que pour de petits messages (plus petits que 10 Ko) la communication entre nœuds est deux fois plus lente qu'une communication interne au nœud. Par ailleurs, la bande passante maximale exploitée pour les communications inter nœuds est seulement de 400 Mo/s, bien en-dessous de la capacité du lien. De même, pour les gros messages (jusqu'à 4 Mo), les auteurs constatent que jusqu'à 500 Ko, les communications au sein du même nœud sont entre 1,5 et 2 fois plus rapides que les communications entre nœuds. Au-delà, les performances se tassent et les auteurs constatent une différence de 200 Mo/s seulement, la bande passante maximale atteinte étant de 900 Mo/s. Même si cette performance est déjà bien meilleure que pour les petits messages, on reste moins performant qu'avec des communications purement locales et toujours incapable d'exploiter pleinement le lien Infiniband. On constate ainsi que les communications ont un coût important.

D'autant plus que, une fois que les auteurs ont optimisé leur réseau de communication entre les nœuds, ils n'obtiennent dans le meilleur des cas qu'un gain performance que de 25X en utilisant 256 cœurs de calcul, dans le meilleur des cas. On est donc très loin d'un gain linéaire. On constate d'ailleurs que le gain de performance maximum qui puisse être atteint dans un programme qui s'appuie au maximum sur des communications est de l'ordre de $\sqrt{nombre\ de\ coeurs}$ (Rajasekaran & Reif, 2007).

De plus, dans le cas d'une application parallèle, la problématique des communications interprocessus ou celle des exclusions mutuelles se pose. En effet, lors du développement d'une application parallèle, il peut arriver que plusieurs processus ou threads doivent accéder à la même ressource, soit pour la modifier, soit simplement pour la lire. Que se passe-t-il dans le cas où un processus modifie la variable tandis qu'un autre est en train de la lire ou de l'écrire ? Ces scenarii sont connus sous le nom de « *race condition* » (condition de compétition) et peuvent conduire à des résultats désastreux : crash de l'application, comportement imprévisible, voire problème de sécurité. Il convient donc de s'assurer que deux processus ne peuvent entrer en compétition sur une ressource, alors nommée ressource critique. Le problème a été en premier lieu formalisé dans (Dijkstra, 1965) où une première solution a été proposée : l'exclusion mutuelle (en anglais, « *mutual exclusion* », ou encore « mutex »). Plus tard, des améliorations à la solution initiale ont été proposées dans (Knuth, 1966) (De Bruijn, 1967) (Eisenberg & McGuire, 1972) (Lamport, 1974). Une solution alternative, le sémaphore, a été proposée par (Dijkstra, 1968). Dans tous les cas, le principe de base reste le même, les différents processus vont partager un objet commun, le mutex ou le sémaphore et se placeront en attente sur cet objet. Celui-ci ne libérera qu'un seul processus à la fois pour qu'il puisse accéder à la ressource critique. Ce mécanisme assure donc qu'un seul processus à la fois accède à la ressource, si tant est que tous utilisent bien le même mutex ou le même sémaphore. Il est donc fortement conseillé (Vladimirov & Karpusenko, 2013) de réduire au maximum le nombre de ressources critiques et de préférer la duplication des données quand elles sont en lectures afin de les rendre privées et d'assurer la meilleure efficacité de calcul, comme montré dans (Gourgand & Hill, 1990).

On constate donc ici qu'optimiser une application en la parallélisant implique un travail supplémentaire à faire pour éviter ces conditions de compétition. Par ailleurs, on peut constater qu'à certains moments de l'exécution de l'application, on peut avoir des instances de l'application qui sont simplement en attente d'un mécanisme d'exclusion et qui passent donc l'essentiel de leur à attendre. Cela induit donc évidemment une perte de performance qui limite le gain maximum qui pourrait être théoriquement atteint. De plus, dans certains cas, notamment en cas de mauvaise maîtrise du parallélisme de l'application, des situations d'inter-blocage (« *deadlock* ») peuvent se

produire. Un cas typique étant un processus attendant la libération d'un mutex pour pouvoir libérer le sien, tandis que ce mutex doit être libéré par un autre processus qui lui-même attend la libération du mutex du premier processus pour libérer le sien. Les deux processus attendent tous deux la libération par l'autre processus du mutex pour pouvoir libérer le leur. Comme ceci ne pourra jamais arriver, l'application est alors bloquée indéfiniment jusqu'à ce qu'un opérateur vienne la tuer. De plus, quand la gestion des exclusions est réalisée rigoureusement et ne présente pas de situation d'inter-blocage, il se pose la question de l'ordonnancement des threads et des processus. Arriveront-ils toujours dans le même ordre vers ce mécanisme d'une exécution à l'autre ? Seront-ils toujours libérés dans le même ordre d'une exécution à l'autre ? Ces deux questions soulèvent alors la question de la reproductibilité de l'application. Si on lance l'application dans les mêmes conditions, et avec les mêmes conditions initiales, donne-t-elle les mêmes résultats ?

4.2 – Reproductibilité des résultats

Ainsi, avec l'optimisation se pose immédiatement (ou en tout cas, devrait se poser immédiatement) la question de la reproductibilité des résultats. Dans le cadre d'applications scientifiques, il est nécessaire de pouvoir reproduire les résultats, afin de pouvoir faire valider l'expérience par des pairs. Cette validation par les pairs, qui pourraient reproduire les résultats, est un sujet qui a préoccupé Jon Claerbout pendant une (grande) partie de sa carrière (Claerbout, 1990) (Claerbout & Karrenbach, 1992) (Schwab, Karrenbach, & Claerbout, 2000). Par conséquent si optimiser l'application revient à perdre la reproductibilité des résultats entre la version non-optimisée et la version optimisée, voire entre plusieurs exécutions de la version optimisée, les conditions nécessaires pour pouvoir considérer les résultats de l'application comme des résultats scientifiques ne sont plus satisfaites, et l'optimisation n'apporte donc strictement rien.

Dans (Hellekalek, 1998) l'auteur avertit déjà des dangers de l'optimisation par la parallélisation des applications. Dans cet article, l'auteur s'intéresse aux simulations de Monte Carlo (parallèles). Les simulations de Monte Carlo, qui sont des simulations stochastiques ont besoin d'un flux de nombres aléatoires. Pour permettre la reproductibilité de l'expérience, il est nécessaire d'utiliser un générateur de nombres pseudo-aléatoires (*Pseudo Random Number Generator* ou PRNG). En effet, pour une initialisation donnée d'un PRNG, la séquence de nombres aléatoires générés est toujours la même. La reproductibilité des résultats de simulation nécessite que d'une exécution à l'autre, chaque nombre aléatoire soit utilisé pour la même chose. Par contre lorsque l'on parallélise une simulation de Monte Carlo, la question de la qualité statistique des flux de nombres pseudo-

aléatoires se repose à nouveau. Même si celle-ci n'impacte pas directement la reproductibilité de la simulation (on peut avoir une mauvaise qualité statistique en étant reproductible), il est néanmoins nécessaire d'y prêter attention pour la qualité finale des résultats. Avoir des résultats reproductible mais de qualité insuffisante ne ferait pas grand sens dans le monde scientifique. Dans (Hellekalek, 1998) l'auteur explique que peu de PRNGs de l'époque sont capables de fournir des séquences de nombres pseudo-aléatoires de qualité pour des simulations parallèles. De même, il met en évidence des défauts consécutifs à l'utilisation de séquences de nombres de mauvaise qualité statistique. Aujourd'hui, différentes batteries de tests permettent de vérifier la qualité statistique des PRNGs, la batterie la plus évoluée et la plus complète est proposée par l'équipe de Pierre l'Ecuyer : TestU01 (L'Ecuyer & Simard, 2007), avant cette batterie de tests, les experts utilisaient DIEHARD (Marsaglia, 1996), et historiquement nous trouvons une série de tests initiaux proposés dans (Knuth, 1969). Différentes solutions ont été proposées pour répondre au problème de la qualité statistique des séquences aléatoires utilisées dans les simulations parallèles, *via* l'implémentation de générateurs de nombres pseudo-aléatoires prévus générer des séquences parallèles « indépendantes » (Mascagni & Srinivasan, 2000) (L'Ecuyer, Simard, Chen, & Kelton, 2002) (Brunner, 2003) (Salmon, Moraes, Dror, & Shaw, 2011) (L'Ecuyer, Oreshkin, & Simard, 2014). Les méthodes auxquelles on peut recourir pour obtenir ce résultat peuvent être la génération d'un paramétrage unique (et reproductible) du générateur à chaque instanciation, ou l'utilisation d'une clé unique pour l'initialisation. Ils fournissent alors un flux indépendant pour chaque instanciation et/ou clé. Il est donc possible que chaque entité (processus, thread) parallèle ait son propre flux stochastique « indépendant ».

Un état de l'art des méthodes modernes de distribution de séquences pseudo-aléatoires sur des processeurs classiques et sur des accélérateurs matériels est proposé par (Hill, Mazel, Passerat-Palmbach, & Traore, 2013). Certaines approches de parallélisation sont lourdes à mettre en œuvre, et elles peuvent nécessiter parfois un pré-calcul en amont ainsi que le stockage de données (les statuts initiaux des PRNGs) pour produire des initialisations correctes des différentes instances parallèles des PRNGs. En effet, pour que l'on puisse obtenir plusieurs flux (ou sous-flux) stochastiques qui ne se recouvrent pas, il est important d'appliquer une des méthodes décrites dans (Hill et al., 2013). Ces différentes méthodes ont toutes pour but de permettre de disposer de nombres pseudo-aléatoires pour chaque entité parallèle. Une méthode permettant d'obtenir plusieurs séquences pseudo-aléatoires qui ne se recouvrent pas est la méthode dite de « *sequence splitting* ». Cette méthode requiert d'estimer à l'avance une borne supérieure du nombre de nombres pseudo-aléatoires que pourrait consommer une instance parallèle. Dès lors, grâce à un algorithme de « *jump-ahead* », qui permet de « sauter » d'un statut de PRNG à un autre, sans avoir à

générer les nombres intermédiaires, on peut construire autant de statuts initiaux que d'instances parallèles.

Avec le PRNG Mersenne Twister (Matsumoto & Nishimura, 1998b), deux approches peuvent être utilisées. Mersenne Twister dispose d'un algorithme de « *jump-ahead* » (Haramoto, Matsumoto, & L'Ecuyer, 2008). Mais Mersenne Twister peut aussi être paramétré, grâce à « *Dynamic Creator* » (Matsumoto & Nishimura, 1998a). Cet algorithme permet de générer, à la volée de nouveaux paramètres pour le générateur. L'algorithme *Dynamic Creator* est conçu de manière à ce que les séquences pseudo-aléatoires générées avec ces nouveaux paramètres aient de bonnes propriétés d' « indépendance » (i.e. satisfassent, à un bon niveau de confiance, des tests statistiques d'indépendance). On se retrouve ainsi avec plusieurs instances du générateur Mersenne Twister. Avec chacune de ces instances, il est également possible de faire du « *jump-ahead* ». On obtient donc finalement plusieurs flux stochastiques « indépendants » dont on peut extraire des sous-flux stochastiques « indépendants ». Cela permet de distribuer correctement ses flux indépendants entre instances parallèles.

Une autre méthode potentiellement dangereuse à employer avec parcimonie et réflexion, mais qui se retrouve malheureusement trop fréquemment employée est la méthode dite de « *random spacing* ». Celle-ci consiste à initialiser le PRNG, quel qu'il soit, avec un statut aléatoire. Ce statut peut être fourni par le tirage d'un nombre pseudo-aléatoire sur un PRNG secondaire, ou plus fréquemment par le nombre de secondes écoulées depuis 1970 au moment du lancement de l'instance parallèle[23]. Dans ce dernier cas, on ne maîtrise pas du tout à quel endroit du flux stochastique du générateur l'instance a été initialisée et on perd tout reproductibilité. Le fait que les statuts initiaux soient générés aléatoirement revient à placer aléatoirement les sous-séquences dans la séquence principale du PRNG : il y a donc un risque non nul de chevauchement de ces sous-séquences ; dans ce cas, les sous-séquences ne sont plus indépendantes, et les intervalles de confiance calculés sont faux. Dans (Hill et al., 2013), les auteurs font état d'un exemple de mauvaise utilisation avec un générateur qui présente une probabilité de recouvrement supérieure à 99,9%, le générateur était lui-même de faible qualité. Autant dire que cela peut avoir de réelles implications statistiques, en introduisant des biais significatifs dans les résultats, en donnant à des évènements un poids plus importants qu'ils ne devraient avoir en raison de leur répétition

Dans (Urbatsch & Evans, 1999) les auteurs considèrent des résultats comme reproductibles quand, pour diverses exécutions avec les mêmes données en entrée, ceux-ci sont exactement les mêmes, et non simplement statistiquement équivalents. Pour arriver à cette fin, toujours dans le contexte des

[23] Aussi connue comme l'initialisation avec time(NULL)

simulations de Monte Carlo, les auteurs affirment qu'il faut impérativement enlever toute dépendance du code au processeur. La partie parallèle du code (y compris le générateur, donc) doit être indépendante de l'élément de calcul sur lequel elle s'exécute, ce qui permet de s'affranchir des problématiques d'ordonnancement du système d'exploitation : on ne s'appuie pas sur l'ordre d'exécution des threads. Par ailleurs, pour le cas particulier des simulations de Monte Carlo propagation de particules, les auteurs incitent à assigner un identifiant unique à chaque particule et au générateur qui lui est assigné en propre, ceci afin de permettre de retrouver toujours le même résultat pour une initialisation donnée de la simulation et de ses PRNGs. Ceci implique notamment, de supprimer toute dépendance entre les particules ou, en général, entre les objets simulés. Néanmoins, dans cet article, les auteurs ne poussent pas encore le concept jusqu'au bout, et ne considèrent pas le problème de la reproductibilité numérique dans le cas d'exécutions parallèles de la simulation. Il est possible d'obtenir ce niveau de reproductibilité, avec des résultats rigoureusement identiques quels que soient les modes d'exécution du programme (séquentiel, parallèle ou distribué) avec la méthode proposée dans (Hill, 2015).

Néanmoins, dans (Urbatsch & Evans, 1999) les auteurs soulèvent un autre point sensible qui peut amener à des problèmes de reproductibilité dans les applications scientifiques, qu'elles soient des simulations de Monte Carlo ou non. Ce que les auteurs définissent comme les « erreurs d'arrondis » ramène à la question de la représentation et de la manipulation des nombres en virgule flottante, utilisés dans de nombreux codes scientifiques.

La représentation des nombres à virgule flottante en informatique est standardisée par le standard IEEE-754 (IEEE, 2008). Ce standard définit notamment comment doivent être représentés les nombres en mémoire, comment les opérations sur ces nombres doivent être réalisées et quels arrondis peuvent être réalisés. L'objectif de ce standard est de permettre à toutes les plates-formes matérielles et logicielles d'utiliser les mêmes méthodes sur les nombres à virgule flottante. Le défaut de l'implémentation de ce standard est que sa précision peut conduire à une lenteur d'exécution de l'application finale. De sorte que, beaucoup de compilateur, de système d'exploitations, voire de matériels fournissent des fonctions qui sont majoritairement dans le cadre de l'IEEE-754 mais qui prennent certaines libertés pour accélérer les traitements. Il avait été ainsi reporté dans (Gustavson, Moreira, & Enenkel, 1999) des problèmes dans les calculs de FMA (*Fused Multiply-Add*) de Java. S'ils étaient reproductibles d'une exécution à l'autre, quelle que soit la machine, ils étaient constamment faux.

La problématique de reproductibilité vis-à-vis des nombres à virgules flottantes est donc multifactorielle. Une absence de reproductibilité entre deux exécutions peut provenir de différences entre les architectures matérielles utilisées, mais également de l'utilisation de compilateurs

différents ou d'un simple changement dans les paramètres de compilation. Par défaut, les compilateurs tels qu'ICC (Intel), MSVC (Microsoft) ou GCC (GNU) vont compiler le code sans appliquer strictement le standard IEEE-754, l'objectif étant d'obtenir le meilleur compromis entre performances et précision de l'application (avec un risque accepté de non reproductibilité). Les compilateurs donnent néanmoins des paramètres de compilation qui permettent de moduler l'application du standard jusqu'à un point où l'exécutable généré sera totalement conforme. Dans (Corden & Kreitzer, 2014), les auteurs expliquent notamment l'ensemble des paramètres de ICC et leur impact sur l'application du standard. Dans (Fleegal, 2004), on retrouve les mêmes recommandations pour MSVC.

Dans certaines des publications présentées dans la section 3, des lignes de compilation sont données et l'on peut constater le peu d'importance accordée à la reproductibilité numérique, certains auteurs compilant même leur simulation en mode « *fast maths* » qui est généralement la pire option de compilation trouvable sur un compilateur en qui concerne la reproductibilité numérique. En effet, celle-ci, pour maximiser les performances, fait les pires approximations possibles. On est alors sûr d'avoir des résultats différents d'une architecture matérielle à l'autre, voire d'un compilateur à l'autre et dans les pires cas, d'une exécution à l'autre, en raison de problématiques d'alignement (Rosenquist, 2012). Lorsque les résultats finaux sont des images et non des valeurs numériques, ce type d'optimisation est tout à fait concevable. Lorsqu'il s'agit de maintenir une reproductibilité numérique, elles ne sont plus envisageables et on découvre que la reproductibilité a un coût. Dans (Rosenquist, 2012) l'auteur annonce une perte de performance de l'ordre de 12% à 15% pour conserver la reproductibilité sur les nombres à virgule flottante avec IEEE-754. Dans (Vladimirov & Karpusenko, 2013) les auteurs donnent l'exemple d'une application qui fait une simple boucle sur des calculs en virgule flottante impliquant l'utilisation de la fonction mathématique racine carrée. L'une est compilée avec les optimisations numériques d'ICC les plus agressives (« `-fpmodel fast=2` »), l'autre avec celles qui sont un peu moins agressives (« `-fp-model fast=1` »), mais toujours approximatives. Aux alentours de la 23 000ème itération, ils obtiennent déjà une dérive dans les résultats. A la fin de l'exécution, à la 45 000ème itération, ils constatent une différence significative dans les résultats, bien au-delà de la précision attendue avec le type « double » dont ils font usage. En revanche, le temps d'exécution est divisé par deux.

Enfin, dans (Intel, 2014) les auteurs expliquent qu'entre un processeur Intel et un Xeon Phi, dans tous les cas, il se peut qu'il n'y ait pas de « *bit-for-bit identical results* »[24], c'est-à-dire de résultats

[24] « In general, floating-point computations on an Intel Xeon Phi coprocessor may not give bit-for-bit identical results to the equivalent computations on an Intel Xeon processor, even though underlying hardware instructions conform to the same standards. »

identiques au bit près. Même si l'exécutable est compilé avec la plus stricte application possible du standard IEEE-754 sur les deux plates-formes, il se peut qu'il soit impossible d'avoir le même résultat au bit près. Les auteurs donnent alors des instructions de compilations pour réduire cette différence, comme désactiver la FMA matérielle du Xeon Phi, qui n'existe pas dans un processeur classique. Mais en aucun cas, ils ne garantissent la stricte reproductibilité des résultats, bien que le matériel, lui, soit toujours conforme au standard IEEE-754.

5 – Conclusions

Dans ce chapitre, nous nous sommes intéressés à l'optimisation d'une application de calcul scientifique. Dans un premier temps, nous avons présenté de nombreux outils permettant le profilage de l'application. Ces outils permettent d'analyser l'application et d'en extraire les informations sur ses performances et sur les endroits où l'application pourrait bénéficier d'optimisations. Nous avons constaté que les outils sont nombreux et variés, notamment par la nature de leur mise en œuvre, mais également par les compteurs de performance qu'ils surveillent. Ainsi, pour avoir la meilleure vision des performances d'une application, il peut être nécessaire d'utiliser plusieurs outils de profilage.

Cela a permis de dégager deux axes majeurs pour l'optimisation d'une application. Le premier axe consiste à optimiser l'aspect purement calculatoire de l'application. Nous avons d'abord exposé quelques méthodes simples d'optimisation qui sont rapides et peu coûteuses à mettre en œuvre pour obtenir des gains de performances qui peuvent aller jusqu'à 5X. Puis, nous nous sommes intéressés à un meilleur usage du processeur, notamment, de ses instructions vectorielles. Celles-ci permettent d'appliquer un même traitement à plusieurs données simultanément. Elles permettent donc un gain de performance qui peut être significatif. Néanmoins, leur utilisation est moins évidente, et peut requérir des changements profonds dans le code, voire l'utilisation de bibliothèques externes dédiées quand le compilateur n'est plus capable de vectoriser le code correctement. Ces instructions vectorielles apportent des gains de performances bien plus importants que les optimisations précédentes. Certaines bibliothèques permettent donc d'obtenir des gains de performances de l'ordre de 8X. Nous avons ensuite présenté des optimisations inter-procédures. Celles-ci ne requièrent aucune intervention sur le code source, uniquement des changements dans la compilation, voire dans les outils de compilation. Les gains de performances, ici, sont à nouveau plus limités, inférieurs à 2X. Enfin, nous avons présenté deux mécanismes qui permettent d'exploiter au mieux un processeur ou un cluster : la parallélisation ainsi que la

distribution des calculs. La première est le simple fait d'exécuter en parallèle certaines sections du code, voire la majeure partie du code. Celle-ci reste locale à la machine. Quand on exécute plusieurs instances de la même application sur la même machine ou sur plusieurs machines, il est question de distribution. Afin de permettre l'exploitation de ces deux mécanismes, nous avons présenté des outils tels que MPI ou OpenMP qui permettent de faciliter l'implémentation dans l'application.

Ensuite, nous nous sommes intéressés au second axe majeur pour l'optimisation : la bonne gestion de la mémoire de l'application. Nous avons en effet expliqué qu'une machine dispose de plusieurs types de mémoire de taille et de vitesse d'accès inversement proportionnelles. Il convient donc de les utiliser avec parcimonie et réflexion. Ici, les bonnes utilisations sont extrêmement variées à mettre en œuvre, passant d'une redéfinition des données en mémoire pour les agencer autrement, aux techniques d'espacement correctes des données pour que le processeur puisse y accéder le plus efficacement possible. Bien maîtriser l'utilisation de la mémoire de l'application peut apporter des gains de performances supérieurs à 3X. Néanmoins, certaines tentatives d'optimisation peuvent s'avérer contre productives et amener à des pertes de performances de l'ordre de 0,75X. Il est donc nécessaire d'être prudent dans ces mises en œuvre et de faire confiance au compilateur.

Enfin, nous avons présenté une dernière solution pour l'optimisation d'une application qui passe par le changement de matériel, et notamment l'utilisation d'accélérateurs matériels dédiés. Nous nous sommes particulièrement intéressés au dernier produit d'Intel, le Xeon Phi qui est au cœur de notre travail de thèse. Nous avons présenté différentes études le mettant en balance avec les GP-GPUs d'NVIDIA. Ainsi, il a été observé qu'une fois correctement utilisé, un Xeon Phi peut permettre d'obtenir des performances équivalentes aux derniers GP-GPUs voire, dans certaines conditions, permettre d'obtenir de meilleurs performances. Néanmoins, leur utilisation n'est pas immédiate, et un minimum de travail est à fournir pour porter l'application sur Xeon Phi. S'il est moindre que celui nécessaire pour les GP-GPUs, il se révèle souvent indispensable, faute de quoi, les performances de l'application sur Xeon Phi ne seront pas au rendez-vous. Les articles cités précédemment montrent que les gains de performances par rapport à un processeur correctement exploité avec un Xeon Phi sont souvent de l'ordre de 2X à 3X.

Une fois ces différentes méthodes d'optimisations présentées, nous avons pointé deux limites majeures à l'optimisation des codes. Ainsi, dans le cas où on la parallélise, l'optimisation d'une application peut être fortement réduit, en raison des problématiques de communication entre les différentes instances, mais également en raison du partage des ressources critiques qui va provoquer des temps d'inactivité, le temps que celles-ci soient disponibles. Par ailleurs, le simple fait que l'ensemble de l'application ne puisse être parallélisée ou distribuée a un impact sur les performances maximales que l'on pourra obtenir d'elle.

La seconde limite que nous avons pointée se situe dans le prix à payer pour avoir une application optimisée. L'optimisation d'une application peut se faire au dépend de la qualité numérique ou statistique de ses résultats, voire au prix de la perte de reproductibilité de ceux-ci. On perdrait alors la possibilité de répéter l'expérience scientifique, que ce soit sur une autre machine, voire sur la même machine. Ce risque est à prendre en considération si l'on souhaite rester dans le cadre de la méthode scientifique, et toutes les optimisations, notamment numériques ne sont pas bonnes à prendre. C'est la raison pour laquelle, elles n'ont, d'ailleurs, pas été présentées dans la section 3. Le fait de vouloir conserver la reproductibilité des résultats, et par conséquent de ne pas exploiter ces optimisations numériques, a donc un coût en termes de performance de l'ordre de 12% à 15% selon les applications.

Dans ce chapitre, nous avons ainsi présenté les différentes façons d'optimiser une application une fois qu'elle a été profilée, depuis les optimisations les plus simples, jusqu'au changement d'architecture. Nous les avons mises en perspective pour nous éviter des complications liées à la qualité et à la reproductibilité des résultats lors de l'application de ces méthodes d'optimisations. Les applications de ces méthodes seront présentées dans les chapitres suivants.

Enfin, nous insistons sur le fait que, parfois, la meilleure optimisation consiste, en faisant preuve d'un minimum de réflexion, à utiliser des outils bien adaptés à l'application et à l'architecture matérielle sur laquelle elle est destinée à être exécutée. C'est ainsi que dans (Drake, 2014) l'auteur constate avec une certaine ironie que, en réfléchissant juste à une problématique d'analyse de fichiers et d'extractions de données, il avait été capable, en utilisant sa machine personnelle et la ligne de commande Linux, d'obtenir un gain de performance de 235X par rapport à une infrastructure de type « cloud » reposant sur `Hadoop` (Shvachko, Kuang, Radia, & Chansler, 2010).

Chapitre 3 : Propositions

1 – Introduction

Lors des chapitres précédents, nous avons pu poser le contexte de travail pour notre thèse. A travers ce chapitre, nous allons exposer la toute première simulation de Monte Carlo de l'expérience ToMuVol (`tomusim`), ses limites et nos propositions pour l'améliorer, tant d'un point de vue performance que rigueur d'implémentation stochastique. Ensuite, nous proposerons une méthode pour porter une simulation de Monte Carlo sur Intel Xeon Phi. Par la suite, nous proposerons une méthode pour réduire la consommation mémoire d'une simulation de Monte Carlo parallélisée. Nous proposerons ensuite une approche du profilage d'application parallèle grâce à l'aspect. Enfin, nous proposerons une stratégie pour développer une simulation de Monte Carlo parallèle et numériquement reproductible.
Les propositions exposées dans ce chapitre seront mises en pratique dans les deux chapitres suivant.

2 – Optimisation d'une simulation de Monte Carlo : `tomusim`

2.1 – Présentation de `tomusim`

Le logiciel `tomusim` a été développé comme un outil spécialisé de simulation de Monte Carlo en physique des particules. Son objectif est uniquement de permettre de simuler la propagation des muons à travers une cible. Cette simulation inclut également, en préambule, la génération pseudo-aléatoire des muons qui seront par la suite propagés. Dans les premières versions de `tomusim`, celui-ci simulait également un détecteur simplifié à l'issue de la propagation. Cet objectif de simulation a été suivi afin de permettre de mimer un phénomène réel et naturel qui intéresse l'expérience ToMuVol : la propagation de muons à travers un grand édifice, ici, le Puy-de-Dôme. Ceci permet notamment de vérifier que l'on comprend bien ce que l'on observe avec le détecteur ToMuVol. Les observables physiques mesurées sont-elles bien prédites par la simulation ? Sinon, la description des processus physiques devrait être améliorée dans la simulation pour reproduire ce qui est observé. Ainsi, en paramètres d'entrée, `tomusim` prend notamment un intervalle d'énergies, et un modèle de cible qui permet de décrire le milieu sur lequel on va travailler. En sortie, `tomusim` va donner bien plus d'informations qu'un détecteur. Là où le détecteur ne peut fournir que la transmission de muons qui l'a traversé, la simulation peut fournir bien plus

d'informations, telles que les positions exactes des muons ainsi que leurs énergies à différentes étapes de la propagation, ou encore la densité moyenne au long de la trajectoire parcourue par la particule lors de sa propagation. Cependant, il est possible de retraiter les données *a posteriori* pour récupérer des données similaires à celles du détecteur.

Le logiciel a été, à l'origine, développé par Felix Fehr (Fehr, 2012). Son objectif est de réaliser des simulations de Monte Carlo de la propagation de muons dans une cible de grande taille : dans notre cas, le Puy-de-Dôme. Sa sortie est un fichier binaire pour le logiciel de traitement de grands volumes de données, `ROOT` (Brun & Rademakers, 1997), qui contient toutes les informations sur chaque muon. Le logiciel peut être configuré à partir d'un fichier texte ou en ligne de commandes. Par défaut, le logiciel dispose de sa propre configuration, purement générique. Pour pouvoir lancer des simulations correctement et obtenir des résultats cohérents, celle-ci devra forcément être retouchée.

Le programme n'embarque pas de modèle de cible sur lequel effectuer la propagation et sans modèle, il ne démarre pas. Le modèle doit donc être fourni en paramètre au lancement, sous la forme d'un fichier binaire pour la bibliothèque `jHepWork` (Chekanov, 2008). La génération du fichier binaire peut être réalisée de plusieurs façons, grâce à des outils annexes. Un de ceux-ci permet de convertir un fichier texte qui contient les coordonnées des points d'un cube contenant la cible ainsi que la densité en chaque point vers un format de sortie binaire, que ce soit le format binaire de `ROOT` ou de `jHepWork`. Un second outil permet de convertir un fichier binaire du logiciel `ROOT`, contenant la géométrie de la cible, vers ce type de fichier texte. Enfin, un troisième outil permet de générer facilement le fichier binaire `ROOT`, ainsi que le fichier texte associé, avec une géométrie dedans, qui aura été décrite dans un programme `ROOT`, à l'aide des instructions de création de géométrie disponibles dans `ROOT`. Ce dernier programme permet donc de générer n'importe quelle géométrie. Enfin, comme il n'existe pas de programme de conversion directe de `ROOT` à `jHepWork`, il est nécessaire de passer par le format texte intermédiaire. Un cas d'usage typique sera donc de modifier le programme `ROOT` qui permet de générer la géométrie en fichier texte afin de parfaitement décrire sa géométrie puis, de convertir le fichier texte ainsi généré en fichier `jHepWork` pour ensuite pouvoir lancer la simulation avec ce modèle de géométrie.

Dans un première partie de ce chapitre, nous allons tout d'abord décrire le modèle de propagation qui a été utilisé pour implémenter la propagation des particules dans `tomusim`. Puis, dans une seconde partie, nous décrirons les différentes briques logicielles utilisées pour l'implémentation de la simulation, ainsi que leurs interactions. Ensuite, nous exposerons les différentes optimisations réalisées dans un premier temps et, nous détaillerons les algorithmes qui ont été implémentés dans

`tomusim` suite à ces optimisations. Enfin, nous exposerons les gains de performance obtenus une fois que la simulation a été distribuée, ainsi que les limites qui ont été mises en évidence.

2.2 – Structure du programme

2.2.1 – Muon Monte Carlo (MMC)

Le cœur du programme, qui s'occupe de la propagation des muons était en réalité un programme récupéré de l'expérience AMANDA (Hulth, 1996), appelé *Muon Monte Carlo*, soit MMC (Chirkin & Rhode, 2001). MMC est écrit en Java et s'appuie notamment sur `jHepWork`, d'où l'utilisation d'objets binaires pour stocker la géométrie de la cible sous forme voxélisée, c'est-à-dire discrétisée en *voxels* (contraction de « volumetric pixel »), qui modélisent des cubes de 10 mètres de côtés, et qui stockent la densité au milieu du *voxel*, repéré par ses trois coordonnées spatiales, x, y et z. Pour pouvoir utiliser MMC dans le cadre de `tomusim`, il a fallu implémenter un modèle « Tomuvol3D » qui simule la propagation des muons dans un milieu rocheux non homogène (qui est donc la cible). Par ailleurs, étant donné que l'interface du modèle était initialement faite pour `tomusim`, il a été possible de modifier cette interface pour Tomuvol3D et de la moduler selon nos besoins. Ainsi, ce modèle permet d'avoir plus d'informations sur les propagations simulées, contrairement à la situation réelle, comme le point d'entrée et le point de sortie de la cible.
On notera qu'à l'origine, MMC utilisait un générateur de nombres pseudo-aléatoires de type congruentiel linéaire, ou LCG, pour *Linear Congruential Generator* (Knuth, 1969). Ces générateurs comportent des défauts structurels et ne devraient plus être utilisés pour des applications scientifiques (L'Ecuyer, 2010).

2.2.2 – ROOT

Afin de gérer l'ensemble des données produites dans `tomusim`, ainsi que les processus physiques et pseudo-aléatoires, comme la génération aléatoire des particules (qui seront ensuite données à MMC pour qu'il les propage) `tomusim` fait appel à ROOT. Ce logiciel fournit des fonctions mathématiques plus élaborées et plus complètes que celles fournies par défaut sur Linux. Il est par exemple possible de manipuler directement des histogrammes représentatifs d'intégrales de fonctions. ROOT fournit également un ensemble de fonctions qui facilitent le stockage de données dans des structures adéquates, telles que les histogrammes que nous avons mentionnés. Ce sont ces

structures qui sont écrites dans le fichier binaire de sortie. L'utilisation de ROOT permet donc de réduire fortement les difficultés liées à la programmation pour pouvoir se concentrer sur la tâche principale : la Physique.

A l'origine écrit en C++, un « *binding* » Python est fourni pour pouvoir utiliser ROOT dans un programme écrit en Python : pyROOT.

2.2.3 – tomusim – brique principale

Le reste du programme tomusim est écrit dans le langage de programmation Python. Le choix de ce langage est lié à sa simplicité et à sa rapidité de mise en œuvre (Van Rossum, 2010). L'interfaçage avec les autres parties décrites plus haut se fait à l'aide de pyROOT pour ROOT et avec l'entrée et la sortie standard pour MMC.

Les interactions entre les différentes briques logicielles de tomusim sont schématisées sur la figure 1.

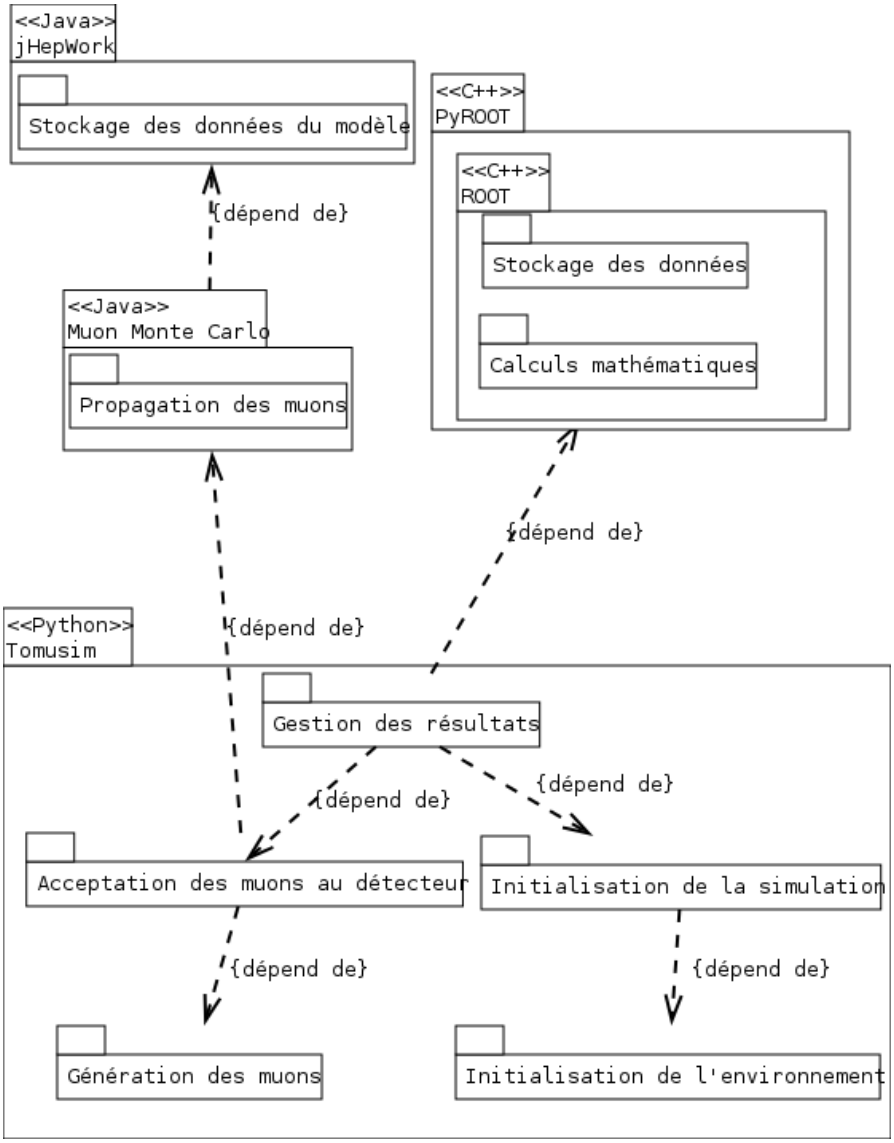

Figure 3-1 : Interactions entre les briques logicielles de `tomusim`

Il y est bien mis en avant le rôle de supervision de `tomusim`. S'il initie la simulation, et fournit les particules à MMC, il ne gère pas la propagation qui est confiée en intégralité à MMC.

Il est à noter que les muons sont générés aléatoirement dans `tomusim`. Pour ce faire, `tomusim` utilise un générateur de nombres pseudo-aléatoires, Mersenne Twister (ou MT) (Matsumoto & Nishimura, 1998b), implémenté dans ROOT. Cette implémentation correspond à celle fournie par ses créateurs eux-mêmes sur leur site web. MT est initialisé au démarrage grâce à la technique de « *Random Spacing* » (Hill et al., 2013). En effet, lorsque `tomusim` est démarré, il faut lui fournir un entier, qui servira de graine (ou « *seed* ») à un générateur Ranlux (Lüscher, 1994) qui tire deux nombres pseudo-aléatoires sont tirés, un pour initialiser MT, et l'autre pour initialiser le LCG qui sera utilisé par MMC. Ce mélange de générateur de structures et de qualités différentes montre une des limites du développement de logiciel scientifiques par des non-spécialistes.

2.3 – Limites et propositions

Dans la structure du programme, on peut pointer plusieurs sujets à discussion et à amélioration. Dans un premier temps, la multiplicité des programmes et des langages de programmation utilisés rend `tomusim` peu efficace, celui-ci perdant énormément de temps dans des échanges d'un programme à l'autre. En effet, si les langages sont différents, les représentations mémoires des données sont différentes. Mais, pire encore, le transfert de données entre deux programmes différents est rendu plus difficile par le fait qu'ils ne partagent rien. Par ailleurs, communiquer *via* les interfaces standards est particulièrement coûteux et peu efficace (communication par chaînes de caractères). Utiliser un langage commun à plusieurs briques logicielles, tel que le C++, était nécessaire.

Des résultats mettant en évidence la lenteur des interfaces ont notamment été obtenus grâce à un profilage de l'application (profilage *a posteriori* – c'est-à-dire, après le passage en C++) avec le logiciel `valgrind` (Nethercote & Seward, 2003) : 35% du temps d'exécution de `tomusim` était perdu dans la conversion des nombres à virgule flottante en chaîne de caractères et 11% du temps était perdu dans la conversion inverse. Il était donc critique de se débarrasser de cette méthode de communication lente (et peu efficace) pour utiliser une méthode réellement fonctionnelle, et possible en C++, comme l'utilisation de la *Java Native Interface* (JNI).

Par exemple, il fallait environ 16h30, sur un processeur Intel Xeon X5650[25], pour obtenir, au niveau du détecteur inclus dans `tomusim`, 10 000 particules acceptées (qui ont donc encore une énergie non nulle lorsqu'elles arrivent sur le détecteur), d'énergie initiale comprise entre 1 GeV (Giga Électronvolt) et 50 TeV (Tera Électronvolt). Dans la réalité, pour capturer autant de particules, selon les données expérimentales, le détecteur de ToMuVol aurait besoin de 15h. Ceci signifie que notre simulation était plus lente que le processus physique qu'elle reproduisait. Par ailleurs, pour avoir de bonnes statistiques, il est nécessaire d'avoir un mois de prise de vue, ce qui aurait correspondu à plus d'un mois de simulation. De plus, pour pouvoir reconstruire le modèle du Puy-de-Dôme, il faudrait itérer sur le modèle, c'est-à-dire, partir d'un modèle simple et le compléter au fur et à mesure des simulations, et générer plusieurs fois un mois de prise de vue, dans le but d'affiner celui-ci. Ceci n'était pas envisageable.

Enfin, la gestion des nombres aléatoires dans `tomusim` était problématique. Tout d'abord, MMC et `tomusim` utilisaient trois générateurs de nombres pseudo-aléatoires, alors que pour cette application, une gestion rigoureuse des flux stochastiques n'en aurait nécessité que deux. De plus,

[25] http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI

l'un de ces trois générateurs, de type LCG, n'est pas du tout recommandé pour les simulations de Monte Carlo (L'Ecuyer, 2010) car il est de faible qualité statistique et, à cause de sa trop faible période, il pose la question du risque de chevauchement des flux stochastiques utilisés par les différentes réplications. En effet, les intervalles de confiance fournis par une simulation de Monte Carlo ne sont exacts que si les flux stochastiques utilisés par les différentes réplications sont indépendants, donc, *a fortiori*, disjoints. Or, l'utilisation conjointe de générateurs de nombres pseudoaléatoires ayant une faible période et de la technique de parallélisation- des flux dite de *random spacing* fait courir un risque non négligeable de chevauchement des flux stochastiques (Hill et al., 2013), et par conséquent de biais statistiques dans les résultats, du fait de la non indépendance des flux. Nous avons donc proposé de restreindre le nombre de flux stochastiques, en en utilisant un unique, partagé entre `tomusim` et MMC, dont l'initialisation soit reproductible et fiable.

Ces améliorations et leurs effets seront décrits en détails dans le chapitre 4.

3 – Portage d'une application sur Intel Xeon Phi

L'objectif annoncé d'Intel lors de la commercialisation du Xeon Phi était de dire qu'il suffirait de prendre une application, de la compiler sur Xeon Phi pour pouvoir exploiter la carte accélératrice et d'en tirer déjà un gain de performance. L'état de l'art a permis d'établir que le gain de performance n'était pas ni si évident ni si immédiat en cas de simple compilation. De plus, dans la majorité des cas, la compilation elle-même n'est pas immédiate et requiert un peu de travail en amont. En effet, contrairement à la compilation d'une application pour une plate-forme de calcul classique, ici, on parle de compilation croisée (ou *cross-compilation*). Il n'est pas possible de compiler l'application directement sur Xeon Phi pour Xeon Phi. Il va falloir compiler sur une autre machine, avec une architecture tierce mais en ciblant le Xeon Phi pour l'exécution du code. Cette compilation croisée pour Xeon Phi peut devenir complexe selon les cas.

3.1 – Technique de compilation croisée d'une simulation sur pour Xeon Phi

Nous allons considérer la compilation de la simulation avec le compilateur ICC d'Intel. Lors de l'installation de la pile logicielle nécessaire à l'utilisation du Xeon Phi sur un nœud de calcul, Intel fournit gracieusement une compilation propriétaire de GCC, qui est capable de produire des binaires pour la plate-forme Xeon Phi. Néanmoins, celle-ci ne sert qu'à la compilation des outils systèmes pour Xeon Phi et ne présente pas d'optimisations particulières, contrairement à ICC.

Pour effectuer une compilation croisée avec ICC, il n'est pas question d'utiliser une chaîne de compilation annexe, comme la pratique courante le veut (d'où la présence d'ailleurs de cette chaîne de compilation basée sur GCC pour Xeon Phi dans les outils Intel). Avec ICC, il suffit dd'utiliser le même binaire, et de rajouter l'option de compilation « `-mmic` ». Malheureusement, si cela semble faciliter le processus comparativement à la chaîne de compilation croisée, cela le complique en réalité, par exemple, avec la méthode de compilation s'appuyant sur les outils GNU : `autotools`, `configure`. Ces outils ont été prévus pour la compilation croisée grâce à une chaîne de compilation croisée et grâce aux arguments « `--build` » et « `--host` », qui permettent respectivement de définir les architectures cible et hôte. L'outil `configure` s'appuie notamment sur des vérifications de la machine cible pour pouvoir compiler l'application. Il va compiler de petits fragments de code, qu'il va tenter d'exécuter sur la machine pour vérifier le bon fonctionnement du compilateur, des en-têtes, voire récupérer des constantes du système. Dans le cadre de la compilation croisée, il ne peut donc pas exécuter les applications de test qu'il a compilées, puisqu'elles ne sont pas prévues pour l'architecture hôte. L'utilisation des deux arguments permet donc de signaler la compilation croisée à l'outil `configure` pour qu'il ne fasse que des vérifications minimales et ne tente pas d'exécuter des binaires qui ne pourraient pas l'être.

Dans le cas de la compilation croisée avec ICC, pour les exécutables avec `configure`, il suffit de lui changer des variables d'environnement. CC et/ou CXX pour fournir le chemin vers les exécutables de ICC et CFLAGS et/ou CXXFLAGS pour fournir l'argument « `-mmic` ». Ceci fait, `configure` n'aura aucune information concernant la compilation croisée, et celle-ci échouera. Une solution que nous avons trouvé pour pouvoir effectuer une compilation croisée avec ICC est de rajouter l'argument « `--host=x86_64-unknown-linux-gnu` » lors de l'appel à l'outil `configure`. Dès lors, l'hôte sur lequel `configure` s'exécute étant inconnu, celui-ci ne tentera pas d'exécuter ses applications de test, et vérifiera simplement la bonne compilation, comme pour une compilation croisée. Sur le code 1, se trouve un exemple de compilation de la bibliothèque de compression `xz`. Celle-ci s'appuie sur configure pour la compilation.

```
env CC=/opt/intel/bin/icc CFLAGS="-static-intel -mmic" \
../xz-5.0.3_src/configure --prefix=/physics_phi/xz-5.0.3_prefix/ \
--host=x86_64-unknown-linux-gnu
make -j32
make install
```

Code 3-1 : Compilation croisée de la bibliothèque `xz` pour Xeon Phi

Une autre stratégie de compilation s'appuie sur `CMake` (Martin, Hoffman, Cedilnik, King, & Nuendorf, 2010). La compilation croisée avec `CMake` est beaucoup plus simple. En effet, dans la majorité des cas, la compilation avec `CMake` consiste simplement à lancer `CMake` qui vérifiera la présence des compilateurs et générera les fichiers `makefile`. Il suffit alors de lancer `make`, et la compilation débute. Il n'y a pas d'étape où `CMake` tente de compiler des fragments de code pour vérifier le bon fonctionnement de la chaîne de compilation ou les retours du compilateur. Cela implique qu'il suffit de fournir à la ligne de commande `CMake` les arguments -DCMAKE_CXX_COMPILER et/ou -DCMAKE_C_COMPILER pour fournir le chemin vers les exécutables de ICC et les arguments -DCMAKE_CXX_FLAGS et/ou -DCMAKE_C_FLAGS pour y ajouter « `-mmic` ». La compilation devrait alors se dérouler sans accroc comme le montre le Code 2 qui permet de compiler la bibliothèque scientifique `CLHEP` (Lönnblad, 1994). Evidemment, là, il n'est pas question d'intercaler un « `make tests` » qui échouerait de façon certaine, en raison de la compilation croisée, faute de pouvoir lancer les exécutables de test.

```
cmake -DCMAKE_CXX_COMPILER=/opt/intel/bin/icpc \
-DCMAKE_CXX_FLAGS="-mmic -static-intel" \
-DCMAKE_C_COMPILER=/opt/intel/bin/icc \
-DCMAKE_C_FLAGS="-mmic -static-intel" \
-DCMAKE_EXE_LINKER_FLAGS="-mmic -static-intel" \
-DCMAKE_INSTALL_PREFIX=/physics_phi/clhep-2.1.3.1_prefix \
-DCMAKE_MODULE_LINKER_FLAGS="-mmic -static-intel" \
-DCMAKE_SHARED_LINKER_FLAGS="-mmic -static-intel" \
../clhep-2.1.3.1_src/CLHEP/
make -j32
make install
```

Code 3-2 : Compilation croisée de la bibliothèque `CLEHP` pour Xeon Phi

Néanmoins, il est des cas où la compilation croisée peut s'avérer plus complexe. Notamment les cas où, durant la compilation, un exécutable va être créé et celui-ci va servir à générer des fichiers quelconques qui seront eux-mêmes compilés dans le reste du processus de compilation. Le problème, dans le cas de la compilation croisée est que cet exécutable, faisant partie du processus de compilation, a été compilé pour l'architecture cible et non pour l'architecture hôte : il ne peut donc pas s'exécuter sur celle-ci. Que ce soit avec `configure` ou `CMake`, on s'aperçoit bien ici que le problème ne vient pas de l'outil de configuration de la compilation, mais du processus de compilation lui-même, puisque cela intervient en cours de compilation.

Pour expliquer ce cas et la méthode à employer, nous allons nous appuyer sur le logiciel ROOT (Brun & Rademakers, 1997). Lors de sa compilation, ROOT va générer un ensemble de fichiers qui seront compilés ultérieurement. Ceux-ci sont utilisés pour permettre une fonctionnalité très particulière de ROOT : le logiciel est capable de compiler du C++ à la volée dans un terminal, ou même dans un programme. Il génère donc des fichiers « dictionnaire » qui permettent de décrire les classes connues, non pas seulement de la bibliothèque C++ standard, mais également des classes fournies dans ROOT. On se retrouve donc avec un exécutable qui va être compilé, qui va générer ces fichiers sources qui seront compilés à leur tour. Il n'existe pas de méthode simple pour désactiver cette fonctionnalité dans ROOT tant elle est ancrée dans le logiciel. Il est alors plus simple de recourir à la compilation croisée, même si celle-ci sera particulière.

Dans le cadre de ROOT, nous avons dans un premier temps fait l'inventaire des exécutables utilisés pour générer les fichiers sources. Ils sont au nombre de quatre : « cint_tmp », « rootcint_tmp », « mktypes » et « rlibmap ». La compilation croisée se déroule dès lors en deux étapes. A la première étape, seuls ces quatre exécutables vont être compilés. Cette première étape est une compilation native. Elle utilise ICC, mais sans l'option de compilation « -mmic », cela signifie que les exécutables peuvent être directement utilisés sur l'architecture hôte. Cette première étape est reproduite sur le Code 3.

```
cmake \
-DCMAKE_CXX_COMPILER=/opt/intel/bin/icpc -DCMAKE_CXX_FLAGS="-static-intel"
\
-DCMAKE_C_COMPILER=/opt/intel/bin/icc -DCMAKE_C_FLAGS="-static-intel" \
-DCMAKE_EXE_LINKER_FLAGS="-static-intel" \
-DCMAKE_MODULE_LINKER_FLAGS="-static-intel" \
-DCMAKE_SHARED_LINKER_FLAGS="-static-intel" \
../root-5.34.03-dev_src
make cint_tmp rootcint_tmp mktypes rlibmap
mkdir -p ../build_bin
cp bin/cint_tmp ../build_bin
cp bin/rootcint_tmp ../build_bin
cp bin/mktypes ../build_bin
cp bin/rlibmap ../build_bin
rm -rf *
```

Code 3-3 : Compilation native des outils intermédiaires nécessaires pour la compilation de ROOT

On peut constater sur le code 3 que seuls ces quatre exécutables sont compilés, puis sauvegardés dans un autre répertoire, créé pour l'occasion. Dès lors, tous les autres fichiers sont supprimés pour

permettre la véritable compilation croisée, c'est-à-dire la seconde étape du processus de compilation. Pour cette seconde étape, nous avons cependant dû modifier légèrement les fichiers `CMake` de `ROOT`. En effet, telle qu'était pensée la compilation, `CMake` générait la configuration locale, compilait les exécutables temporaires et les exécutait, compilait l'ensemble puis supprimait ces exécutables intermédiaires. Il n'était donc pas prévu que ces exécutables puissent être indépendants et extérieurs au processus de compilation. Nous avons donc modifié les fichiers `CMake` pour qu'ils prennent en compte une variable booléenne de configuration « USE_EXT_CINT » qui une fois passée sur « ON » désactive la compilation de ces exécutables intermédiaire. Ceux disponibles dans le « PATH » sont alors utilisés à la place.

```
PATH=/physics_phi/build_bin:$PATH cmake -DUSE_EXT_CINT=ON \
-DCMAKE_CXX_COMPILER=/opt/intel/bin/icpc -DCMAKE_CXX_FLAGS="-mmic -static-
intel" \
-DCMAKE_C_COMPILER=/opt/intel/bin/icc -DCMAKE_C_FLAGS="-mmic -static-intel"
\
-DCMAKE_EXE_LINKER_FLAGS="-mmic -static-intel" \
-DCMAKE_INSTALL_PREFIX=/physics_phi/root-5.34.03_prefix \
-DCMAKE_MODULE_LINKER_FLAGS="-mmic -static-intel" \
-DCMAKE_SHARED_LINKER_FLAGS="-mmic -static-intel" \
-Dbuiltin_pcre=OFF -Dbuiltin_zlib=ON -Dbuiltin_lzma=OFF -Dx11=OFF -Dxft=OFF
-Dalien=OFF \
-Dbonjour=OFF -Dcastor=OFF -Dchirp=OFF -Ddcache=OFF -Dfftw3=OFF -
Dfitsio=OFF -Dgviz=OFF \
-Dgfal=OFF -Dglite=OFF -Dhdfs=OFF -Dkrb5=OFF -Dldap=OFF -Dmathmore=OFF -
Dminuit2=ON \
-Dmonalisa=OFF -Dmysql=OFF -Dodbc=OFF -Dopengl=OFF -Doracle=OFF -Dpgsql=OFF
\
-Dpythia6=OFF -Dpythia8=OFF -Drfio=OFF -Dsapdb=OFF -Dshared=OFF -Dsrp=OFF -
Dssl=OFF \
-Dxrootd=OFF -Dpython=OFF \
-DLZMA_INCLUDE_DIR=/physics_phi/xz-5.0.3_prefix/include/ \
-DLZMA_LIBRARY=/physics_phi/xz-5.0.3_prefix/lib/liblzma.so \
-DFREETYPE_INCLUDE_DIR_freetype2=/physics_phi/freetype-2.5-
dev_prefix/include/freetype2/ \
-DFREETYPE_INCLUDE_DIR_ft2build=/physics_phi/freetype-2.5-
dev_prefix/include/freetype2/ \
-DFREETYPE_LIBRARY=/physics_phi/freetype-2.5-dev_prefix/lib/libfreetype.a \
-DPCRE_CONFIG_EXECUTABLE=/physics_phi/pcre-8.31_prefix/pcre-config \
-DLIBXML2_INCLUDE_DIR=/physics_phi/libxml2-2.9.1_prefix/include/libxml2/ \
```

```
-DLIBXML2_LIBRARIES=/physics_phi/libxml2-2.9.1_prefix/lib/libxml2.so \
../root-5.34.03-dev_src
PATH=/physics_phi/build_bin:$PATH make -j32
make install
```

Code 3-4 : Compilation croisée de ROOT (seconde étape)

Le Code 4 couvre l'intégralité des commandes nécessaires pour la compilation croisée de `ROOT` pour Xeon Phi. On y note un point important : la compilation croisée d'un logiciel tel que `ROOT`, qui possède de nombreuses dépendances peut s'avérer complexe. Il faut en effet, non seulement gérer le cas de la compilation croisée du logiciel lui-même, mais également de toutes les dépendances dont il a besoin. Il faut donc choisir avec soin les fonctionnalités que l'on active dans le logiciel. Typiquement, dans le cadre de `ROOT`, on peut constater que le paramètre « `-Dx11=OFF` » est donné. Celui-ci permet de désactiver toutes les fonctionnalités graphiques de `ROOT`. Le Xeon Phi n'a pas de serveur X, et n'est pas prévu pour faire de l'affichage.

3.2 – Optimisation de la simulation pour Xeon Phi

Si les architectures x86 et x86_64 peuvent aujourd'hui manipuler n'importe quelle zone mémoire sans coût majeur, tel n'est pas le cas pour l'architecture `k1om` du Xeon Phi. Comme expliqué dans (S. Li, 2013), il est nécessaire d'aligner les adresses des espaces mémoires que le Xeon Phi va manipuler sur des adresses multiples de 64 bits. Ceci concerne toutes les adresses mémoires : autant celles de tableaux ou d'objets alloués par le système grâce aux appels systèmes adéquats, mais également celles des objets statiques du programme qui pourraient être référencés.

Le compilateur d'Intel (mais également GCC) fournit plusieurs outils qui permettent d'aligner correctement les adresses mémoires. Pour tous les objets statiques, il est possible d'utiliser le mot clé « `__attribute__(aligned(64))` » qui permet de spécifier au compilateur qu'il doit aligner l'adresse sur 64 bits lors de la création de l'objet. Nous avons pu néanmoins constater que lorsque la cible de compilation est le Xeon Phi, le compilateur ICC va tenter d'aligner lui-même correctement les objets statiques.

Lors de l'allocation dynamique de mémoire, il faut utiliser des fonctions intrinsèques du compilateur telles que « `_mm_malloc()` » ou « `_aligned_malloc()` ». Celles-ci prennent en premier argument une taille de mémoire qui sera allouée, comme un appel à `malloc()` standard, et en second argument, l'alignement à respecter. Par exemple « `_mm_alloc(, 64)` » alignera l'allocation sur 64 bits. A noter que la norme POSIX fournit également un appel système qui permet d'allouer de la mémoire dont l'adresse est alignée sur une taille

spécifiée : « `posix_memalign()` ». Toutes ces méthodes permettent donc d'aligner correctement la mémoire, mais dans le cas d'une allocation dynamique, le compilateur est incapable de traquer quelles zones mémoires ont été correctement alignées ou non. Ainsi, dès qu'il est question d'utiliser une zone mémoire, il faut préciser, lors de son utilisation, si celle-ci a été alignée ou non, ceci avec le mot clé « `__assume_aligned(, 64);` ». On donne un exemple de l'utilisation de tous ces mots clés dans le Code 5.

```
int a[100] __attribute__ ((aligned(64)));
void f(int * b, int * c, int n) {
    __assume_aligned(b, 64);
    __assume_aligned(c, 64);
    for (int i = 0; i < min(n, 100); ++i) {
        c[i] = a[i] + b[i];
    }
}
void g(void) {
    int * b = _mm_malloc(100 * sizeof(int), 64);
    int * c = _mm_malloc(100 * sizeof(int), 64);
    f(b, c, 100);
}
```

Code 3-5 : Exemple d'utilisation des mots clés pour l'alignement de la mémoire

Ceci soulève néanmoins un problème quand on programme en C++ : il n'existe pas d'allocateur par défaut qui aligne correctement la mémoire, ce qui force à en implémenter un soi-même au risque d'être moins performant que l'allocateur par défaut qui n'aligne pas la mémoire (mais qui peut venir avec d'autres optimisations). Il n'existe pas non plus de structure de données qui garantisse que, quand on insère des données, celles-ci sont correctement alignées.

Dans tous les cas, si l'on veut exploiter au mieux le Xeon Phi, et en obtenir les meilleures performances, il faut optimiser les accès mémoires, et donc, une simple compilation croisée vers le Xeon Phi ne suffit plus. Ceci implique de revoir le code source et d'ajouter toutes les indications nécessaires pour le compilateur. Non seulement celles décrites plus haut pour les problématiques d'alignement mémoire, mais également celles décrites dans le chapitre de l'état de l'art, qui permettent d'aider le compilateur à savoir quelles boucles peuvent (ou non) être vectorisées. Cette tâche peut s'avérer particulièrement ardue dans les cas où l'application repose sur de nombreuses dépendances et où celles-ci sont importantes en termes de volumétrie de code. On pensera ici

notamment aux simulations de Monte Carlo développées avec `Geant4` (Bogdanov et al., 2006), qui n'est pas optimisé pour Xeon Phi et qui représente environ deux millions de lignes de code. Or une simulation de Monte Carlo développée avec `Geant4` dépendra en majeure partie de `Geant4` et n'aura donc que peu de code « en propre », la majorité de ce dernier consistant en la description de la simulation, la simulation, quant à elle, étant effectuée par `Geant4`.

On peut mettre l'ensemble en perspective : nous avons pu constater que pour les allocations statiques, le compilateur aligne correctement la mémoire sur 64 bits. Par ailleurs, la bibliothèque C standard (`libc`) suit la norme C89 (et C99) qui demande à ce que la mémoire allouée soit alignée correctement pour n'importe quel type « *built-in* » (`char`, `short`, `int`, `long`, `longlong`). L'alignement correct est celui qui convient à la plate-forme cible. Nous avons donc fait le test sur Xeon Phi d'allouer 9 999 zones mémoires d'une taille variable comprise entre 1 octet et 10 000 octets, sans jamais les libérer (afin d'exploiter le tas au maximum). Puis, nous avons vérifié les 9 999 adresses retournées : elles étaient toutes alignées sur 64 bits. *A priori*, le portage d'Intel de la `libc` pour Xeon Phi semble suivre C89. Cependant, comme mis en avant dans (Bernaschi et al., 2014), il faut s'assurer de l'alignement des données auxquelles on accède, et il est donc important de faire usage des mots clés `__assume_aligned` présenté dans le Code 5, mais aussi d'`__assume` qui permet d'indiquer d'autres hypothèses (par exemple sur l'indice d'une boucle) au compilateur.

3.3 – Utilisation du Xeon Phi pour les simulations *memory-bound*

Néanmoins, même sans optimisations particulières, le Xeon Phi peut devenir particulièrement utile pour un certain type d'applications : les applications *memory-bound* (Wulf & McKee, 1995). La première problématique des applications *memory-bound* est qu'elles ne peuvent pas accéder assez rapidement à leurs données en mémoires et passent la majeure partie du temps à les attendre, plutôt qu'à les traiter.

Si le Xeon Phi a peu de mémoire (entre 8 et 16 Go), il possède en revanche une bande-passante bien plus importante que celle fournie par les meilleurs processeurs actuels. La bande-passante d'un processeur actuel est de l'ordre de 60 à 70 Gbits/s, pour les hauts de gamme les plus performants tandis que celle d'un Xeon Phi est entre 320 Gbits/s et 350 Gbits/s. Pour une application *memory-bound*, le Xeon Phi présente donc un intérêt architectural, même sans optimisation. Il peut alors permettre des gains de performance substantiels.

Nous avons ainsi fait le test avec une simulation de Monte Carlo, développée avec `Geant4`, qui était fortement *memory-bound*. Nous avons pu nous apercevoir qu'en tentant de la distribuer sur un

nœud de calcul équipé d'un processeur Intel Xeon E5-2687W[26], le gain de performance maximal qu'il était possible d'obtenir était à peine plus que 2X, et ce malgré l'utilisation complète de la machine pour le calcul en distribuant sur tous les cœurs logiques. La figure 2 montre la courbe de gain de performance quand l'application est distribuée sur le processeur Xeon E5. On y voit clairement le palier atteint.

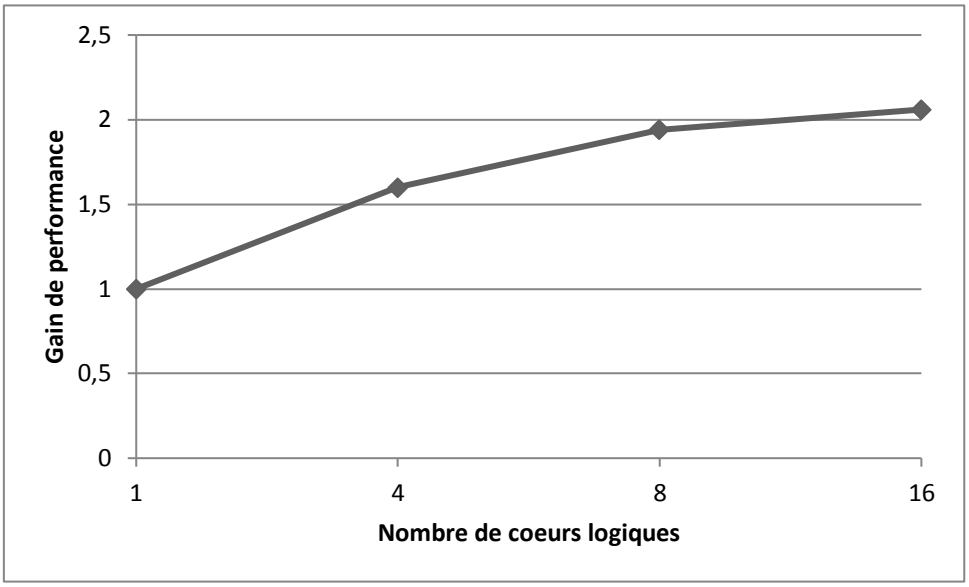

Figure 3-2 : Gain de performance lors de la distribution de la simulation sur les cœurs logiques du Xeon E5-2687W

En revanche, quand cette même simulation est distribuée de la même façon sur Xeon Phi 7120P[27], on n'observe plus aucun phénomène de palier (figure 3). Le gain de performance est aux environs de 150X. On constate par ailleurs que pour les 60 premiers threads matériels, la courbe est réellement proche de la courbe identité (courbe sombre).

[26] http://ark.intel.com/products/64582/Intel-Xeon-Processor-E5-2687W-20M-Cache-3_10-GHz-8_00-GTs-Intel-QPI

[27] http://ark.intel.com/products/75799/Intel-Xeon-Phi-Coprocessor-7120P-16GB-1_238-GHz-61-core

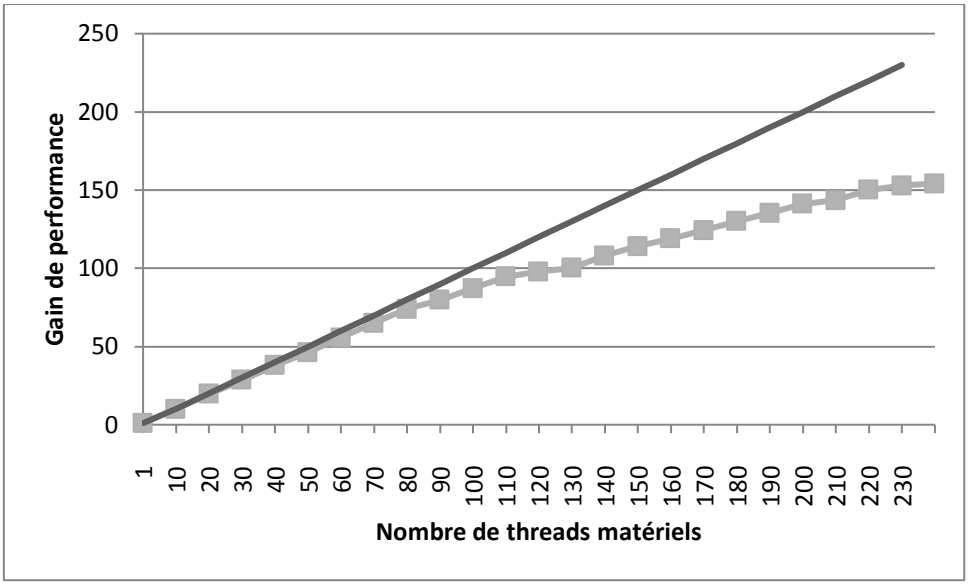

Figure 3-3 : Gain de performance lors de la distribution de la simulation sur les threads matériels du Xeon Phi 7120P

De même, nous avons pu comparer les performances d'une même simulation une fois distribuée sur ces deux architectures, en exploitant le maximum de threads/cœurs logiques possible. Nous avons alors obtenu un gain de performance de 2,56X en faveur du Xeon Phi : là où dans le meilleur cas, le CPU a besoin de plus de 2 heures 15 minutes de calcul, le Xeon Phi se contente, dans le meilleur cas également, de 53 minutes de calcul. Cet effondrement des performances sur le CPU donne clairement l'avantage au Xeon Phi, d'autant plus que la simulation n'a pas été optimisée pour être exécutée sur Xeon Phi, et que les accès mémoires sont laissés à la discrétion du compilateur et du système d'exploitation. On obtient donc réellement dans ce cas précis, un gain de performance important et immédiat, simplement en utilisant la compilation croisée pour exécuter la simulation sur Xeon Phi. Ainsi, à la lumière de ces résultats, nous préconisons l'usage dès que possible du Xeon Phi pour les applications qui seraient « *memory-bound* » : alors que le processeur classique n'arrive pas à fournir des performances suffisantes, une simple compilation croisée sur Xeon Phi permet un gain de performance important. Ainsi, pour un coût d'ingénierie réduit, un simple changement de matériel permettait d'avoir un gain de performance de 2,56X sur les architectures que nous avons testées.

3.4 – Economie de mémoire grâce à KSM

Comme nous l'avons évoqué précédemment, le Xeon Phi embarque très peu de mémoire RAM (maximum 16 Go pour le 7120P). Nous avons notamment fait l'expérience, avec la simulation précédente de la distribuer sur l'ensemble des threads matériels d'un Xeon Phi 5110P. Cela s'est

révélé impossible : le Xeon Phi a fini par crasher, faute de mémoire suffisante. Dans le cas où la simulation est distribuée, et donc place en mémoire les mêmes données, il est possible de partager les données entre les différentes instances qui s'exécutent en parallèle grâce à KSM (Kernel Shared Memory) (Arcangeli, Eidus, & Wright, 2009). KSM avait été à l'origine pensé dans le cas de la virtualisation : on place plusieurs machines, généralement avec le même système d'exploitation et le même noyau sur un même hyperviseur. On se retrouve donc avec des zones mémoires qui contiennent exactement la même chose, et qui pourtant sont dupliquées et consomment inutilement de la mémoire. L'objectif de KSM est donc de permettre de fusionner les pages mémoires qui contiennent exactement la même chose, si tant est que l'utilisateur l'ait autorisé. Cette technologie a fait son apparition dans Linux 2.6.32. Le Xeon Phi, quant à lui, utilise une version modifiée de Linux 2.6.38. Malgré les modifications, celle-ci embarque toujours KSM. Il est donc tout à fait possible d'utiliser KSM sur Xeon Phi, sous réserve de l'activer dans le noyau.

Par ailleurs, deux mots clés sont importants pour l'utilisation de KSM : « autorisé » et « page ». KSM ne peut pas fusionner n'importe quelles zones mémoires au hasard, sous réserve qu'elles contiennent les mêmes données. KSM ne peut fusionner que des portions de pages, à la condition qu'elles soient placées en début de page et qu'elles contiennent les mêmes données sur la même longueur. Il convient donc de changer la méthode d'allocation de la mémoire pour que les zones que l'on veut voir « *fusionnables* » soient toutes au début d'une page mémoire. Par ailleurs, il faut autoriser la fusion desdites pages sur la bonne longueur, sinon KSM les ignorera purement et simplement, même si elles contiennent des données identiques, pour des raisons de sécurité et de performance, notamment. On voit donc qu'un travail de *refactoring* du code pourrait être important.

Il existe cependant un projet « `ksm_preload` »[28] qui permet de s'affranchir de cette contrainte. Il s'agit d'une bibliothèque qui s'utilise grâce à la variable d'environnement « LD_PRELOAD ». Dans ce cas, elle sera chargée avant le lancement de l'application, et même avant la résolution des liens de l'application. Elle peut donc se substituer aux appels systèmes classiques type « `malloc()` » etc. Son implémentation prend alors le dessus, pour évaluer si, en fonction de sa taille, une zone mémoire présente un intérêt à être fusionnée. Si tel est le cas, l'allocation aura lieu en début de page et sera rendue *fusionnable*. Le développeur n'a donc aucun travail à faire, si ce n'est de modifier le démarrage de l'application.

Nous avons fait le test sur notre simulation. La figure 4 montre la distribution de la simulation sur un Xeon Phi 5110P, avec 120 instances en parallèle, sans KSM. La figure 5 montre la même distribution, mais avec KSM activé et avec l'utilisation de `ksm_preload`.

[28] https://github.com/unbrice/ksm_preload

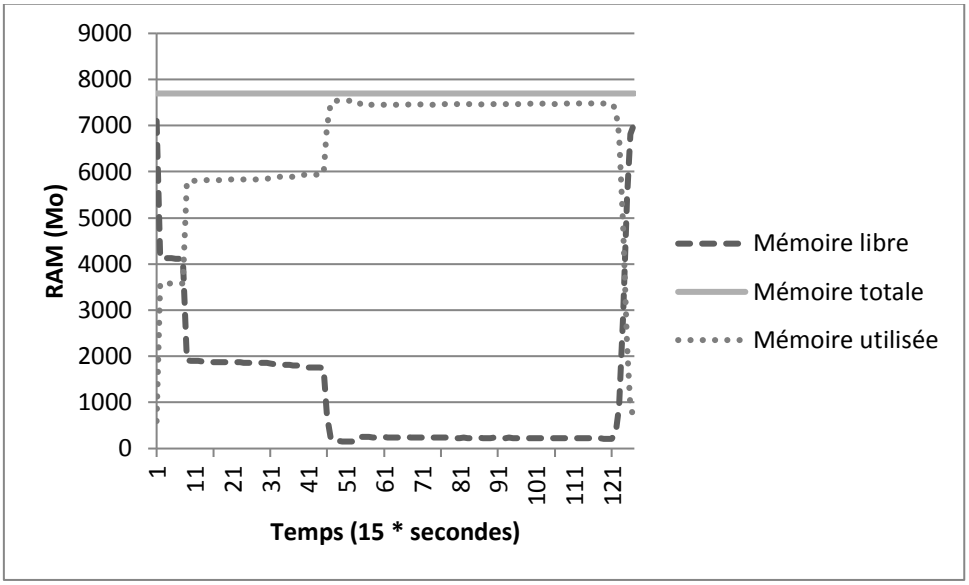

Figure 3-4 : Distribution de 120 instances de la simulation sans KSM sur Xeon Phi 5110P

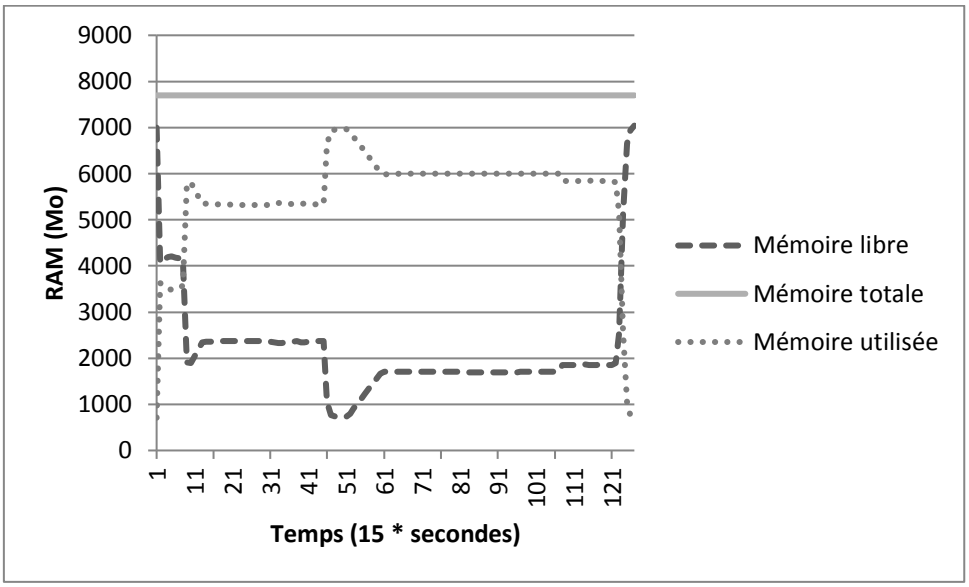

Figure 3-5 : Distribution de 120 instances de la simulation avec KSM sur Xeon Phi 5110P

Nous pouvons constater sur la figure 4 qu'avec 120 instances, sans KSM, l'intégralité de la mémoire disponible du Xeon Phi est consommée, ou qu'en tout cas, il n'en reste pas suffisamment pour faire tenir une nouvelle instance en mémoire. En revanche, on peut constater sur la figure 5, que l'utilisation de KSM a deux avantages : elle libère quasiment un 1 Go de mémoire RAM, et ne provoque pas de ralentissement notable. Nous avons ainsi pu calculer qu'il était possible de faire tourner 20 instances supplémentaires sur le Xeon Phi, grâce à KSM, ce que l'on peut constater sur la figure 6.

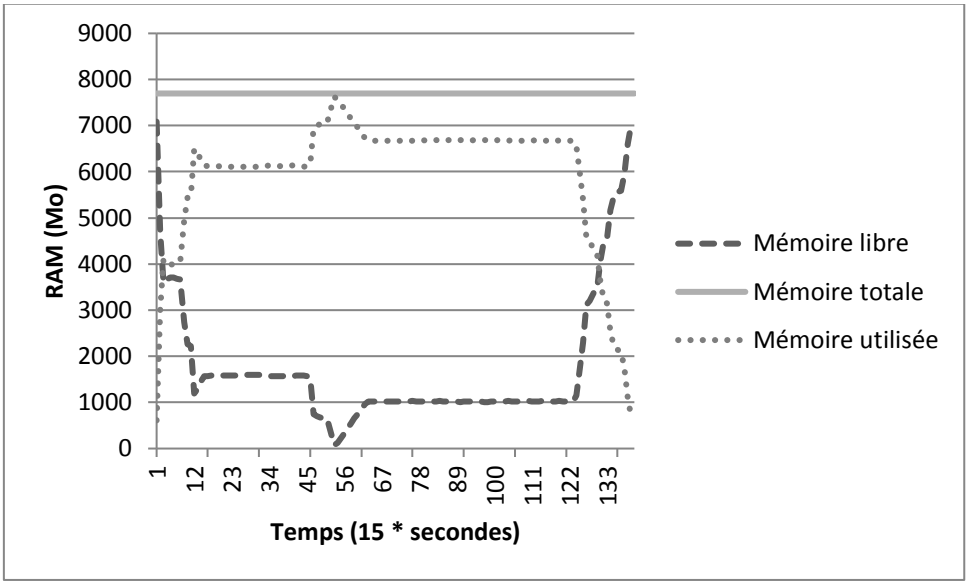

Figure 3-6 : Distribution de 140 instances de la simulation avec KSM sur Xeon Phi 5110P

Nous pouvons également constater, sur la figure 6, que l'ensemble des simulations prend 5 minutes de plus à s'exécuter. Nous avons imputé ce ralentissement à la charge supérieure du Xeon Phi quand il y a plus d'instances en cours d'exécution, ce qui avait déjà été constaté sans utiliser KSM (Schweitzer, Mazel, Cârloganu, & Hill, 2015). Ce n'est donc qu'un comportement normal et attendu.

4 – Développement d'un profileur parallèle en aspect

Comme précisé dans l'état de l'art, certains profileurs notamment en Java, sont écrits grâce au paradigme de programmation dit orientée aspect, comme `CPPROFJ` (Hall, 2002) ou `DJProf` (Pearce et al., 2007). Nous proposerons dans le paragraphe qui suit, une méthode qui permettra de développer un profileur grâce à ce paradigme, et ce, pas nécessairement en Java, à condition d'avoir les bons outils.

Si l'on considère un langage qui supporte l'aspect au niveau de finesse de la fonction (ou qui peut être étendu pour avoir un tel support *via* des outils tiers), cela signifie que l'on pourra avoir une finesse de profilage de l'ordre de la fonction. En effet, avoir une finesse de l'ordre de la fonction signifie que l'on peut avoir un aspect en début en fin d'appel de fonction (*i.e.*, avant l'appel et après l'appel). Le code de l'aspect sera donc exécuté avant l'exécution de la fonction, puis après celle-ci. Cela est important parce que cela permet de collecter des données sur l'état du système, sur l'état de l'application avant l'appel et après l'appel. La différence entre les deux états donne alors le coût consommé par la fonction, qu'il soit en temps de calcul, en consommation mémoire, ou autre.

Deux méthodes de calcul des coûts existent : les méthodes inclusive et exclusive, chacune ayant ses avantages et ses inconvénients. Dans le premier cas, dans le calcul du coût d'une fonction, on prendra en compte le coût de toutes les fonctions filles appelées. Cette méthode présente l'avantage de permettre de cibler très rapidement la branche du programme dans laquelle les coûts sont très élevés pour pouvoir y regarder plus en finesse afin de réduire ces coûts. La deuxième méthode, dite exclusive, permet de ne prendre en compte que les coûts liés à la fonction en propre, les coûts de toutes les fonctions filles étant soustraits du calcul inclusif. Cette méthode présente l'avantage d'avoir le coût réel (et direct) d'une fonction. Dans une branche où l'on sait que le coût est élevé, cela permet de savoir quelles sont les fonctions qui y contribuent fortement. Il est facile de passer d'une méthode à l'autre : le coût inclusif d'une fonction `f` s'obtient en sommant tous les coûts exclusifs des fonctions appelées suite à l'appel de `f` ; le coût exclusif de `f` s'obtient en soustrayant les coûts inclusifs des fonctions filles de `f` au coût inclusif de `f`. Le stockage des coûts peut donc se faire d'une unique façon. Dans le cas de `valgrind` par exemple, les coûts sont stockés de façon exclusive. En revanche, lors de leur mise en forme, `kcachegrind` affiche à la fois les coûts inclusifs et les coûts exclusifs pour permettre à l'utilisateur final d'avoir toutes les données pertinentes. Dans le cas du développement d'un profileur, il s'agit de choisir juste l'une ou l'autre méthode de stockage. Pour rester compatible avec les outils existants (`valgrind` et `kcachegrind`), choisir les coûts exclusifs est nécessaire.

Le développement du profileur avec l'aspect consiste donc à mesurer les coûts des fonctions. Pour pouvoir établir les coûts exclusifs, il est nécessaire de maintenir un arbre contextuel d'appel. Cet arbre n'est là que pour établir la filiation des fonctions. Il est contextuel parce qu'il conserve la trace de la filiation dans le contexte général d'appel. Ainsi, chaque fonction n'a qu'une fonction parente dans le contexte d'appel donné. Il est ainsi possible d'établir une pile d'appels pour une fonction en parcourant l'arbre de la fonction jusqu'à la tête de l'arbre. Et selon la position dans l'arbre, on n'aura pas la même pile d'appels puisque l'entrée est contextuelle. L'arbre fonctionne donc par « niveaux ». Le premier niveau est forcément occupé par la fonction « `main()` », ou de façon générale, par le point d'entrée de l'application, et elle seule. Elle est le point d'entrée du programme. Les fonctions qu'elle appelle directement seront donc au niveau 1. Et les fonctions appelées directement par ces fonctions de niveau 1 seront de niveau 2. Une fonction peut donc apparaître plusieurs fois dans l'arbre, et même plusieurs fois par niveau. Il suffit qu'elle ait plusieurs parents de même niveau ou de niveau différent, donc plusieurs contextes d'appels différents.

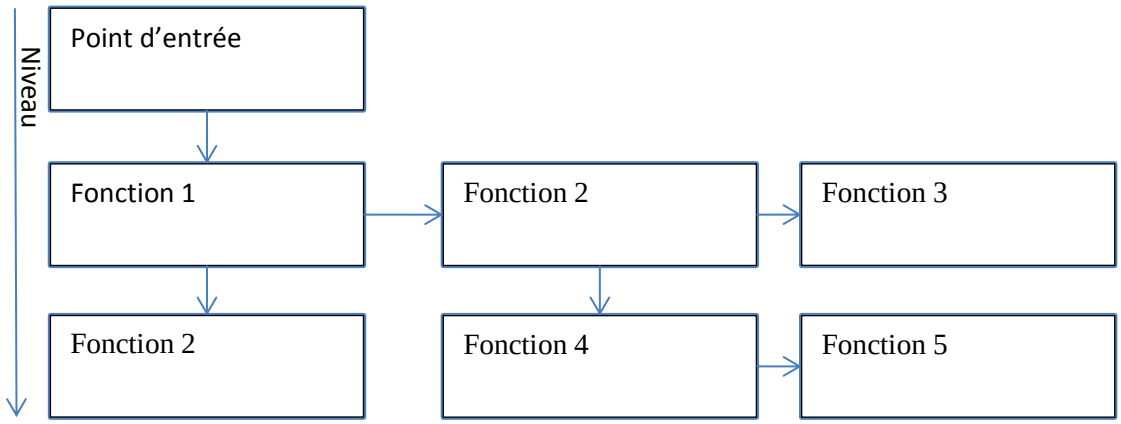

Figure 3-7 : exemple d'arbre contextuel d'appel

Sur la figure 7, on peut voir un exemple d'arbre contextuel d'appel. La fonction point d'entrée va appeler trois fonctions filles : les fonctions 1, 2 et 3. La fonction 1, elle-même, rappellera la fonction 2. Etant donné que le contexte d'appel de la fonction 2 est différent dans les deux cas où elle est appelée, elle apparaît deux fois, à deux niveaux différents. La fonction 2, lorsqu'elle est appelée par le point d'entrée, appelle deux fonctions filles, les fonctions 4 et 5. La fonction 3, quant à elle, n'appelle aucune autre fonction, et constitue une feuille de l'arbre contextuel d'appel.

Les coûts sont donc associés avec les entrées de l'arbre. A la fin du profilage, qui intervient donc à la fin du code de l'aspect exécuté sur la fonction `main()`, il est alors possible de calculer les coûts exclusifs pour chaque fonction étant donné qu'on connait ses coûts inclusifs et l'intégralité des coûts des fonctions filles.

Pour la prise en charge du multithreading, il suffit de rajouter une dimension à l'arbre contextuel d'appel. En effet, pour la version non parallèle, l'arbre possède déjà deux dimensions : les niveaux qui permettent de descendre la pile d'appel et pour chaque niveau, la liste des fonctions qui sont appelées à ce niveau. La troisième dimension consiste donc à rajouter une dimension de profondeur à l'arbre contextuel : à chaque thread, à partir de la fonction où sera créé le thread, on associe une nouvelle version de l'arbre contextuel d'appel, avec comme origine (et donc parent) la fonction qui a créé le thread dans l'arbre original. Ainsi, chaque profondeur correspond à un thread, comme illustré sur la figure 8.

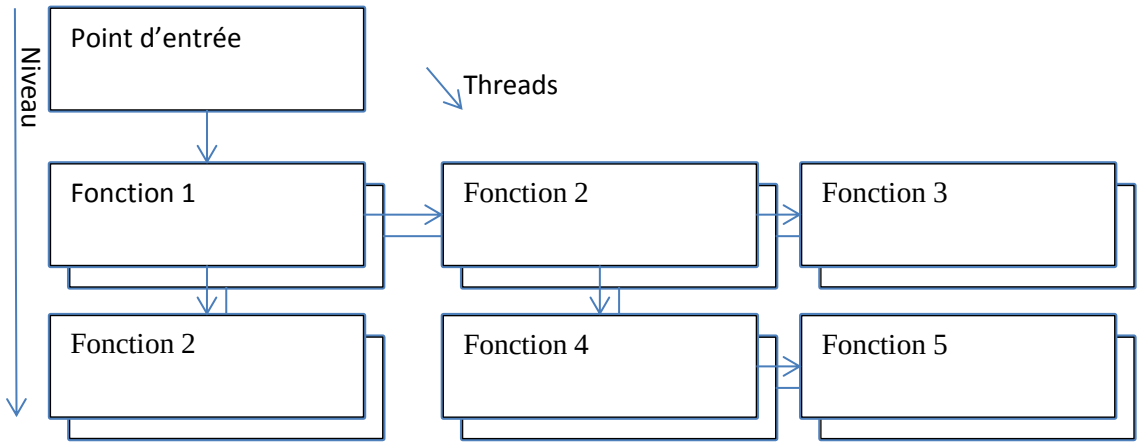

Figure 3-8 : Prise en charge du *multi-threading* dans l'arbre contextuel d'appel

Gérer le parallélisme *via* les *forks* en revanche, ne présente aucun intérêt étant donné qu'un *fork* est un nouveau processus, totalement indépendant du premier.

Ce profileur, développé grâce au paradigme de programmation orientée aspect présente deux avantages majeurs par rapport aux profileurs existants. Le premier et non des moindres : si l'on est capable de de compiler une application sur une architecture matérielle, alors on est automatiquement capable de la profiler vu que ce profileur ne fait qu'ajouter du code au code déjà existant. Cela est particulièrement utile pour les architectures matérielles où les profileurs ne sont pas légion, notamment les Xeon Phi (voir l'état de l'art). Par ailleurs, tous les profileurs (loin de là) ne supportent pas correctement les applications parallèles et induisent soit une corruption des données (`gprof`) soit simulent un parallélisme sur un seul cœur d'exécution (`valgrind`), ou ne prennent en compte que certains appels de fonctions type MPI (voir état de l'art). Dans notre cas, ces limites n'existent pas : le profileur ne change pas le flot d'exécution du programme, ni son parallélisme, ni sa capacité à être porté. Dans le contexte où nous nous situons dans cette thèse : simulations parallèle, utilisation de Xeon Phi, absence d'utilisation d'OpenMP/MPI, ce type de profileur revêt un intérêt non négligeable.

Dans le chapitre suivant, nous implémenterons cette proposition pour créer un profileur C++, en aspect.

5 –Conception d'une simulation stochastique parallèle et numériquement reproductible

5.1 – Règles pour l'implémentation

Une simulation de Monte Carlo étant fortement consommatrice de ressources (CPU, mémoire), il est important qu'elle puisse passer à l'échelle, c'est-à-dire que plus on lui rajoute de CPU ou de RAM, plus la consommation de ces nouvelles ressources permet à la simulation de gagner en performance. Idéalement, si on rajoute un cœur de calcul à une simulation qui en utilisait déjà un, on s'attend à avoir un gain de performance x2. Ce cas reste le cas idéal de la simulation linéaire. Un gain dit super linéaire serait supérieur à x2. Néanmoins, le gain x2 n'est pas toujours atteint. Dans la suite du paragraphe, nous allons proposer une méthode pour mettre au point une simulation qui passe à l'échelle, et ce, dans l'idéal, de façon linéaire.

Par ailleurs, le gain de performance étant totalement inutile s'il induit une perte de reproductibilité, nous placerons la contrainte forte d'avoir une reproductibilité sous les conditions suivantes : changement de paradigme d'exécution, changement d'architecture matérielle.

Notre proposition s'appuie sur le travail de fond réalisé dans (Hill, 2015). Dans cet article, l'auteur donne de nombreuses règles à suivre qui permettent d'avoir une simulation stochastique et parallèle numériquement reproductible. Dans un premier temps, la simulation doit être conçue de façon séquentielle en ayant à l'esprit la parallélisation à venir de l'application. Ceci doit rendre « naturelle » l'utilisation de tous les cœurs de calcul disponibles. Dans un second temps, chaque objet stochastique doit avoir son propre (et unique) flux aléatoire. Cela signifie que l'on a une bijection entre l'objet stochastique et son flux aléatoire. L'un et l'autre sont indissociables et permettent de les identifier mutuellement. Dans un troisième temps, un générateur de nombres pseudo-aléatoires moderne et validé statistiquement doit être utilisé avec une technique rigoureuse de parallélisation de ce générateur.

En plus de ces recommandations, nous nous appuierons sur (Hill et al., 2013) pour ajouter qu'il est nécessaire d'utiliser une méthode de distribution des séquences pseudo-aléatoires aux différents objets stochastiques qui soit totalement reproductible. Une telle méthode permet d'assurer qu'au cours de l'exécution de la simulation, ce sont toujours les mêmes évènements qui sont associés aux mêmes objets. Lorsque l'on dispose d'une borne supérieure du nombre de nombres pseudo-aléatoires utilisés par la simulation, une bonne méthode de distribution des séquences aléatoires est le *sequence splitting*, mais, pour pouvoir l'appliquer, il faut qu'un algorithme de *jump ahead* existe pour le générateur utilisé, pour pouvoir atteindre rapidement (*i.e.*, sans avoir à dérouler toute la séquence pseudo-aléatoire) les statuts initiaux de chaque sous-séquence

On renforcera encore les recommandations sur les générateurs de nombres pseudo-aléatoires (PRNG) en précisant qu'il faut impérativement utiliser un PRNG qui assure la reproductibilité numérique de la séquence, y compris en cas de changement d'architecture. Si ces propriétés ne sont pas vérifiées, la simulation ne sera alors pas numériquement reproductible. On pourra penser qu'il

est évident qu'un PRNG est numériquement reproductible, mais ce n'est pas toujours le cas, ce qui a été montré récemment par notre équipe de recherche au LIMOS (Dao, Maigne, Breton, Nguyen, & Hill, 2014).

De plus, nous faisons le choix de décorréler les entrées/sorties de la simulation du calcul à proprement parler. L'objectif est que l'écriture des résultats (par exemple) n'ait pas d'impact sur les calculs. Nous proposons donc de déporter les écritures vers un thread dédié qui tournera en tâche de fond durant toute la simulation et à qui la simulation transmettra les informations à écrire de façon rapide et peu bloquante.

De façon générale, afin d'avoir une simulation qui passe correctement à l'échelle quand on rajoute des éléments de calcul, il conviendra de garder le nombre de sections critiques au minimum, ainsi que leur durée d'exécution. En cas de besoin, on pourra recourir à la duplication de données plutôt qu'à l'utilisation d'une section critique pour les partager.

Par ailleurs, afin de renforcer l'idée exprimée dans (Hill, 2015) sur la bijection entre flux aléatoire et objet stochastique, nous proposons d'affecter à chaque binôme ainsi constitué, un unique thread de calcul. Ainsi, la durée de vie du thread sera intrinsèquement liée à la durée de vie de l'objet stochastique. Ce choix, outre le fait qu'il permet de séparer proprement les objets stochastiques, permet surtout d'implémenter la simulation séquentiellement avec une anticipation du parallélisme. En effet, si l'on désire exécuter la simulation séquentiellement, il suffit de n'avoir qu'un thread qui s'exécute à la fois. Si l'on désire exécuter la simulation en parallèle, alors il suffit de lancer plusieurs de ces threads en parallèle. Dans le cas présent, on se retrouve proprement dans le modèle de parallélisme, où plusieurs entités non communicantes sont exécutées en parallèle (Fujimoto, 2001).

Afin de pouvoir exécuter la simulation sur plusieurs nœuds de calcul, il est important de pouvoir spécifier, pour chaque instance, à quel « endroit » de l'implémentation séquentielle initiale elle se situe ce qui permet de savoir quels évènements elle a en charge. Cette méthode de « distribution » de la simulation est, par essence, associée avec la technique de *sequence splitting* pour la distribution des flux stochastiques.

Enfin, on prendra un soin particulier à configurer correctement la compilation de la simulation, l'absence de certains paramètres de compilation pouvant conduire à la perte de la reproductibilité numérique.

Nous mettrons en pratique ces recommandations pour l'implémentation d'une simulation de Monte Carlo pour la propagation d'un muon dans un milieu rocheux. Les détails de l'implémentation et l'analyse des résultats (performances et reproductibilité) se trouvent dans le chapitre 5 du présent manuscrit.

5.2 – Analyse de la simulation

Afin d'analyser si la simulation produite est bien hautement parallèle et reproductible, il convient également de regarder proprement son comportement pour éviter des erreurs d'analyse, voire de tirer des conclusions fausses.

Pour l'évaluation des performances dans un premier temps, pour chacune des configurations que l'on cherche à évaluer, on lancera la simulation plusieurs fois de suite avec les mêmes paramètres idéalement entre trois et cinq fois afin d'avoir une bonne vision des performances de la simulation et d'éviter l'aléa induit par le système d'exploitation et le matériel. S'il faut retenir l'un des temps, alors on retiendra le temps d'exécution minimum, qui est représentatif de la vitesse maximale observée de la simulation dans le contexte donné.

Afin de comparer les performances de la simulation entre un processeur et un accélérateur matériel, on prendra notamment soin de régler l'affinité de la simulation sur un seul processeur dans le cas où la machine en possède plusieurs ; pour ce faire, on pourra utiliser la commande `taskset`, par exemple. Car, par défaut, la simulation se répartit sur les cœurs des différents processeurs pour obtenir un maximum de performances (Aas, 2005).

Enfin, pour les mesures de performances l'utilisation de technologies telles que « Turbo » d'Intel est à proscrire, car celles-ci rendent la vitesse du processeur non reproductible, créant des pics de performances. Finalement, ces technologies ne sont pas adaptées pour le calcul à haute performance parce que l'augmentation de performance sur un cœur se fait au détriment d'un autre cœur de calcul.

Pour l'analyse de la reproductibilité des résultats, dans un premier temps, il est important d'affecter un identifiant à chaque objet stochastique, afin de pouvoir facilement les retrouver. Etant donné l'association bijective entre un objet stochastique et son flux aléatoire, nous proposons d'utiliser ke statut initial du générateur comme identifiant. Il sera donc unique et sera l'identification précise de l'objet. Par ailleurs, l'étude doit s'effectuer sur des simulations qui sont suffisamment longues : par exemple, effectuer l'étude sur plusieurs milliers d'évènements permet d'avoir une statistique représentative tandis qu'utiliser une simulation de seulement une dizaine d'évènements serait non concluant, l'échantillon étant trop réduit.

Dans un premier temps, il s'agit de vérifier si, entre deux exécutions, tous les évènements se retrouvent bien dans les mêmes conditions. Il s'agit de vérifier la reproductibilité numérique « *run-to-run* ». Si rien n'est changé entre les deux lancements, la simulation doit donc strictement retourner les mêmes résultats, avec dans l'idéal une reproductibilité numérique « *bit-for-bit* ». Si tel n'est pas le cas, la simulation n'est pas absolument reproductible de façon numérique. Dans le cadre

de simulations à grande échelles et sur des architectures matérielles différentes, des marges d'erreurs relatives minimes peuvent être acceptables (Hill, 2015).

Ensuite, il faudra vérifier si la simulation, lors de l'utilisation du même matériel, retourne les mêmes résultats quel que soit le paradigme d'exécution choisi : parallèle, distribuée sur plusieurs instances, séquentiel. En effet, pour la reproductibilité numérique, le paradigme ne devrait pas avoir d'effets sur les résultats de la simulation. Encore une fois, ici, une reproductibilité numérique « *bit-for-bit* » est attendue étant donné qu'il n'y a pas de changement d'architecture matérielle.

Enfin, pour l'étude de la reproductibilité sur différentes architectures matérielles, il convient de fixer un plan d'expérience facilitant l'étude (par exemple, lancer une unique instance séquentielle facilement configurable). Ensuite, la même simulation avec les mêmes paramètres doit être exécutée sur les deux architectures cibles. Il faut bien évidemment vérifier que les mêmes évènements sont traités sur les deux mêmes architectures, sinon, la simulation ne serait clairement pas reproductible. Normalement, les deux architectures devraient traiter les mêmes évènements grâce à l'utilisation d'une technique reproductible de distribution des flux stochastiques, comme la technique de *sequence splitting*, par exemple.

Selon les architectures étrangères choisies, il est fort probable que la reproductibilité « *bit-for-bit* » soit perdue. Il ne convient donc plus de comparer deux évènements uniquement sur leur représentation binaire. Il faut étudier l'ordre de grandeur de leur différence relative, qui sera représentative de la divergence entre les deux architectures. Pour une valeur donnée (`r`) sur les deux architectures (`arc1` et `arc2`), la différence relative (Δ) s'obtient à l'aide de la formule suivante :

$$\Delta(r_{arc1}, r_{arc2}) = \frac{|r_{arc1} - r_{arc2}|}{Min(|r_{arc1}|, |r_{arc2}|)}.$$

La différence relative peut cependant prendre une valeur extrêmement importante, quand on cherche à étudier une valeur dont la valeur théorique est nulle, non représentative de la reproductibilité de la simulation, ce qui fausse l'analyse de la simulation.

Une fois l'étude de la différence relative faite, il est difficile de conclure sur la (non)-reproductibilité de la simulation car celle-ci dépend de la précision attendue du modèle et de la différence maximale observée. En effet, si la différence maximale est de l'ordre de 10^{-8} mais que le modèle exige une précision de l'ordre de 10^{-10} pour correspondre à la réalité simulée, alors la simulation n'est plus reproductible. En revanche, si on a le même ordre de grandeur de différence mais avec un modèle réel qui est sujet à un bruit de fond important, la simulation se retrouvera tout de même plus précise que la réalité et on pourra considérer qu'elle est numériquement reproductible.

Néanmoins, il faut être prudent avant d'annoncer que nous obtenus une reproductibilité numérique. L'absence de divergence flagrante sur l'échantillon observé ne signifie pas que toute la simulation est reproductible. Il se peut qu'il y ait un ou plusieurs évènements aberrants qui passent inaperçus faute d'avoir été comparés, ou parce qu'ils n'ont pas été rencontrés sur les échantillons simulés. L'intégralité de cette section sera mise en pratique dans le chapitre 5, où seront expliqués en détail l'implémentation, mais également l'analyse des résultats de la simulation. On y constatera que notre simulation est numériquement reproductible, et offre d'excellentes performances.

6 – Au sujet de la vectorisation

Une fois l'application séquentielle correctement optimisée, il devient nécessaire d'exploiter au mieux les différents cœurs de calcul disponibles. Comme expliqué dans le chapitre de l'état de l'art, plusieurs méthodes existent afin de pouvoir paralléliser ou distribuer la simulation. Néanmoins, la parallélisation ne signifie pas nécessairement un gain de performances, bien au contraire. Ainsi, nous avons pu constater que la parallélisation SIMD (*Simple Instruction Multiple Data*) grâce aux instructions vectorielles pouvait conduire à une sévère perte de performance selon le processeur ciblé : processeur Intel Xeon (serveur) ou processeur Intel Core (grand public). Notre proposition dans ce chapitre est donc de ne pas abuser des instructions vectorielles, tant que l'on n'a pas la certitude de leur efficacité réelle sur l'architecture cible.

Pour mettre en évidence ce phénomène, nous avons considéré une simulation de Monte Carlo développée avec `Geant4` (Agostinelli et al., 2003), similaire à la simulation `tomusim` décrite dans (Schweitzer, Mazel, Fehr, Cârloganu, & Hill, 2013). Nous avons fixés tous les paramètres de la simulation ainsi que le statut initial du générateur de nombres pseudo aléatoires, afin d'exécuter toujours la même simulation, avec les mêmes données. Les différences en temps de calcul entre les différentes exécutions viennent uniquement du système d'exploitation (ordonnancement, etc.). Pour chaque test, nous avons lancé la simulation cinq fois de suite, en lançant une simulation témoin et une simulation de test en parallèle. La simulation témoin était lancée à chaque fois pour pouvoir évaluer le temps d'exécution de la simulation dans le contexte de charge immédiat de la machine.

Ce que nous désignons ici par simulation témoin est la simulation de Monte Carlo `tmvg4sim`, implémentée avec `Geant4` qui simule la propagation de muons à travers un modèle simplifié de volcan. Il s'agit de la simulation « `memory-bound` » qui a fait l'objet d'une discussion dans la section 3.3 (puis 3.4). Cette simulation et `Geant4` sont compilés avec les options de compilation par défaut du mode « release » de `Geant4`, c'est-à-dire l'option « `-O2` ». Cette simulation et cette

compilation ne changeront plus et serviront de référence. Elle a été compilée avec GCC 4.4.7 en 64 bits.

Par la suite, nous avons pris différents compilateurs ainsi que différentes versions de chaque compilateur, en recompilant toujours, et `Geant4`, et la simulation, avec les options « `-O2` » et « `-march=native` ». Cette dernière option permet de « coller » la simulation à la machine sur laquelle elle sera compilée (et devrait être exécutée). Cela permet d'exploiter tous les jeux d'instructions du processeur, dans l'espoir de gagner en performances. Par opposition à la simulation témoin, cette simulation recompilée sera appelée simulation « *tuned* ».

Pour notre première comparaison de performances, nous recompilé la simulation avec GCC 4.4.7 et l'option de compilation supplémentaire « `-march=native` » et nous l'avons exécutée sur une machine équipée de quatre processeurs Intel Xeon E5-4620[29]. L'avantage de rajouter cette option est que GCC va pouvoir utiliser plus d'instructions vectorielles. Ce sont celles-ci qui changent d'une génération de processeurs à l'autre, et ce sont elles qui nous intéressent ici. Nous constatons immédiatement que la simulation « *tuned* » est plus lente que la simulation témoin : alors que la durée moyenne d'exécution de la simulation témoin est d'environ 247 secondes, la simulation « *tuned* » prend en moyenne environ 252 secondes. Entre les différentes exécutions de la simulation, on constate un écart qui varie entre 3 et 6 secondes. Si la différence est ici petite (de l'ordre de 2% du temps d'exécution total), on constate tout de même une perte de performance.

Nous avons envisagé la possibilité que le compilateur n'exploite pas correctement les instructions vectorielles, ou ait des algorithmes d'optimisation peu performants sur ces instructions. Nous avons donc décidé d'utiliser une version plus récente du compilateur pour la simulation « *tuned* ». La simulation témoin restant, elle, inchangée. Nous avons donc utilisé GCC 4.8.3, qui était, à l'époque des tests, la dernière version stable de GCC (la branche 4.9 posant encore des problèmes, et la 5 n'existant même pas encore). Nous avons réitéré notre procédure de test, en lançant la simulation témoin en même temps que la simulation « *tuned* » et ceci cinq fois d'affilée.

La machine étant un peu plus chargée pour ce test, nous avons constaté une perte de performance pour la simulation témoin. Celle-ci prend environ 265 secondes en moyenne pour s'exécuter. La simulation « *tuned* » reste toujours plus lente car elle prend en moyenne environ 271 secondes. On observe un écart entre les différentes exécutions, qui varie entre 5 et 6 secondes. On constate donc, que notre hypothèse d'un mauvais support des instructions par le compilateur semble fausse dans la mesure où utiliser une version plus récente de GCC n'a pas amélioré les performances.

Nous avons donc décidé d'utiliser un autre compilateur. Nous nous sommes tournés vers l'infrastructure LLVM (Lattner & Adve, 2004) et plus précisément, le compilateur Clang (Lattner,

[29] http://ark.intel.com/fr/products/64607/Intel-Xeon-Processor-E5-4620-16M-Cache-2_20-GHz-7_20-GTs-Intel-QPI

2008). Nous nous sommes appuyés sur la version 3.4.2 de Clang. Nous avons suivi le même protocole que précédemment. Les exécutions de la simulation témoin et de la simulation « *tuned* » ont respectivement demandé, en moyenne environ 273 secondes et 308 secondes. Entre les différentes exécutions, on constate un écart entre la simulation témoin et la « *tuned* » variant de 34 à 36 secondes. On peut clairement constater que les performances se sont effondrées.

Dans tous les cas, nous constatons que plus nous essayons d'exploiter la machine (*i.e.*, plus nous essayons d'utiliser les instructions vectorielles), plus les performances de la simulation s'effondrent. On constate également, qu'a priori, pour nos simulations, Clang n'est pas le bon compilateur pour gagner en performances. Afin d'étayer nos propos, nous avons considéré le pourcentage d'instructions vectorielles dans l'application témoin, mais également dans une des bibliothèques de `Geant4` les plus sollicitées par la simulation. Cette information a été établie grâce à un profilage de la simulation.

Dans la simulation témoin, approximativement 9,6% des instructions sont vectorielles avec le jeu d'instruction AVX. Pour la bibliothèque témoin, environ 27,3% des instructions sont vectorielles, réparties entre SSE2 et MMX. Avec GCC 4.8, nous avons environ 14,2% des instructions qui sont vectorielles, en grande majorité AVX et quelques MMX. Dans la bibliothèque, on retrouve environ 32,1% des instructions qui sont vectorielles, en majeure partie de l'AVX et un peu de MMX. On voit donc clairement que le changement de version de GCC entraîne une plus grande utilisation des instructions vectorielles, mais que ce sont toujours les mêmes jeux d'instructions qui sont utilisés. Avec LLVM, on constate des changements intéressants : dans la simulation, 9,6% environ des instructions sont vectorielles, avec une répartition sur MMX et AVX. Comparativement avec la simulation témoin, plus d'instructions sont avec AVX. Pour la bibliothèque, on retrouve 30,4% des instructions qui sont vectorielles. Les instructions sont réparties entre SSE2, SSE et MMX. Autant les répartitions étaient proches entre les deux versions de GCC, autant avec LLVM, on voit poindre la différence, jusqu'à l'apparition (surprenante) du jeu d'instructions SSE. Ceci peut expliquer le peu de différences en performance entre les deux versions de GCC, mais le véritable fossé en performance entre GCC et LLVM.

Dès lors, nous avons décidé de pousser les tests plus loin. En effet, la machine de test est une machine déclinée serveur avec des processeurs Intel Xeon qui sont destinés pour une utilisation serveur. Les processeurs Intel modernes, même s'ils sont vendus comme des processeurs CISC (*Complex Instruction Set Computer*) parce qu'ils exposent des jeux d'instructions fournis avec de nombreuses instructions, certaines étant particulièrement complexes (par exemple : gestion du chiffrement AES (Akdemir et al., 2010)), ne sont en réalité que des processeurs RISC (*Reduced Instruction Set Computer*). En effet, le cœur du processeur n'est simplement qu'un RISC, avec des

instructions câblées élémentaires. Les instructions supplémentaires, elles, sont codées grâce aux instructions RISC : on parle d'ailleurs de microcode. Ceci permet donc d'orienter la performance des processeurs en fonction des usages qu'on leur destine. Sur un processeur orienté serveur, on mettra moins d'optimisations dans l'implémentation des fonctions vectorielles que pour un processeur orienté bureau. C'est ce que nous avons voulu vérifier.

Nous avons donc reconsidéré tous nos tests, avec cette fois-ci, une machine équipée d'un processeur de la déclinaison « bureau » : le processeur i7-2600K[30]. Nous avons relancé tous nos tests dessus. Nous avons gardé le même système d'exploitation (Scientific Linux 6), ainsi que les mêmes compilateurs. Nous avons néanmoins, tout recompilé pour pouvoir coller proprement les simulations à la nouvelle machine. Nous avons donc comparé en premier lieu GCC 4.3 témoin et « *tuned* ». La simulation témoin a pris environ 164 secondes en moyenne, tandis que la « *tuned* » s'est contentée de 160 secondes environ, toujours en moyenne. On voit clairement une différence par rapport aux comparaisons précédentes, puisqu'ici, nous avons pu mesurer des écarts de 3 à 4 secondes entre les exécutions de la simulation « *tuned* » et celles de la simulation témoin. Si ces écarts sont faibles, ils sont tout de même significatifs. Nous avons décidé de continuer les tests, d'abord avec GCC 4.8 puis Clang. Avec GCC 4.8, la simulation témoin a nécessité en moyenne 170 secondes environ et la simulation « *tuned* » 169 secondes environ. L'écart s'est clairement resserré, et on note une légère baisse de performance des deux. Nous n'avons pas réussi à expliquer clairement la perte de performance, la machine étant dédiée pour ces tests, aucun autre utilisateur ne venant interférer. Nous avons émis l'hypothèse d'une mise en concurrence sur l'accès aux instructions vectorielles, en effet, comme expliqué précédemment, une fois compilée avec GCC 4.8, la simulation utilise bien plus souvent les instructions vectorielles, ceci implique un stress plus important sur les caches et donc des aller-retours plus fréquents vers la mémoire, ce qui nuit aux performances globales. Nous avions par ailleurs constaté cette perte de performance également sur la première machine de test, mais nous l'avions imputé uniquement à l'augmentation de charge de la machine. Le facteur de concurrence sur les instructions vectorielles étant caché par cette augmentation de charge. Avec Clang, les performances sont toujours mauvaises, mais tout de même meilleures qu'avec le processeur Intel Xeon, en effet, si la simulation témoin tourne en moyenne en 171 secondes environ, la simulation « *tuned* » tourne en 188 secondes environ. On constate un écart entre 16 et 20 secondes (à comparer avec un écart de l'ordre de 30 secondes pour le processeur Intel Xeon).

Grâce à cette étude, nous avons pu constater que la recherche de la performance et de l'optimisation pouvait être contre-productive : en tentant de coller au plus près de la machine notre

[30] http://ark.intel.com/fr/products/52214

simulation grâce à la compilation, nous avons constaté que nous dégradions ses performances au lieu de les améliorer. En effet, en ce qui concerne les instructions vectorielles, nous avons établi que leurs performances ne sont pas égales selon la catégorie de processeur que l'on vise. Cependant, on mettra une nuance sur cette conclusion : il s'agissait, ici, uniquement d'instructions vectorielles issues de la vectorisation automatique du compilateur. Une implémentation manuelle correcte aura toujours de meilleures performances que la vectorisation automatique (Estérie et al., 2014; Falcou & Sérot, 2004) et pourrait mieux exploiter le potentiel, certes moins élevé, des processeurs Intel Xeon.

7 – Conclusions

Ce chapitre contient la majorité de nos propositions pour cette thèse. Certaines, très ciblées, seront mises en œuvre dans les chapitres suivants, parce qu'elles ne concernent qu'une seule application et ceci dans un contexte très précis, peu généralisable.

Nos premières propositions concernent l'optimisation d'une simulation de Monte Carlo, qui avait été prototypée en Python. Bien que ces propositions semblent s'appliquer à un cas très précis, on peut néanmoins constater que certaines peuvent s'appliquer en général. En effet, on conseillera de façon générale, pour faire du calcul à haute performance d'éviter tout langage interprété ou compilé à la volée, pour se concentrer sur les langages compilés. Par ailleurs, les éléments apportés sur la bonne gestion des flux stochastiques, en accord avec l'état de l'art s'appliquent à n'importe quelle simulation de Monte Carlo que l'on veut rigoureuse.

Nos propositions concernent ensuite le portage sur la plate-forme Xeon Phi d'une simulation (ou même de façon générale, d'une application), ainsi que de l'intégralité de ses dépendances. Nous considérons également que le Xeon Phi est une plate-forme de calcul de choix pour les applications *memory-bound*. En effet, comme nous l'avons montré, une application qui plafonne à un gain de performance de l'ordre de 2X une fois distribuée sur tous les cœurs d'un processeur standard peut monter jusqu'à 150X lorsqu'elle exploite tous les threads matériels d'un Xeon Phi, permettant un gain de performance de l'ordre de 2,6X par rapport au processeur standard. Par ailleurs, nous avons également proposé l'utilisation de KSM pour Xeon Phi. En effet, le Xeon Phi comparé à un nœud de calcul classique possède réellement peu de mémoire RAM, et ceci peut s'avérer limitant quand une simulation est parallélisée grâce au lancement de plusieurs instances parallèles. Dans ce cas précis, KSM peut venir aider à réduire l'empreinte mémoire générale de la simulation grâce à une fusion des pages mémoires communes entre les instances.

Nous nous sommes ensuite intéressés à la mise au point d'un profileur grâce à la programmation orientée aspect. Cette méthode permet de développer un profileur synchrone qui est indépendant

de la plate-forme : il suffit de recompiler l'intégralité de l'application et du profileur pour chaque plate-forme. Nous avons proposé une méthode pour qu'il puisse gérer les applications parallèles. Le chapitre 4 contiendra une description d'une implémentation dudit profileur par aspect en C++ qui gère non seulement le parallélisme des applications, mais également le parallélisme de l'architecture sur laquelle l'application est lancée.

Puis, nous sommes arrivés au cœur du travail de la thèse : l'implémentation d'une simulation qui soit parallèle, statistiquement cohérente et numériquement reproductible. Afin d'obtenir ces trois objectifs, nous avons formulé différentes propositions qui couvrent à la fois le parallélisme, l'implémentation de la simulation, et l'utilisation des générateurs de nombres pseudo-aléatoires. En supplément de ces propositions, nous avons expliqué comment analyser la simulation ainsi produite afin de s'assurer qu'elle réponde à l'intégralité des critères définis. Une implémentation fonctionnelle et répondant aux trois critères de cette simulation, appliquée à la Physique des Hautes Energie (simulation de la propagation d'un muon à travers la matière) sera décrite et analysée dans le chapitre 5.

Enfin, nous nous sommes intéressés à une approche de la parallélisation : la vectorisation. Et plutôt que de recommander son usage sans distinction, nous conseillons de modérer l'utilisation des instructions vectorielles. En effet, nous avons pu constater que les performances ne sont bien sûr pas égales entre celles d'un processeur grand public et celles d'un processeur orienté serveur de type Xeon. A un tel point que l'utilisation des mécanismes de vectorisation automatique des compilateurs peut grever les performances plutôt que de les améliorer.

Nous mettrons en œuvre les propositions formulées dans ce chapitre dans les chapitres suivants et nous discuterons également de leur impact.

Chapitre 4 : Applications

1 – Introduction

Dans le cadre de la thèse, plusieurs outils et applications ont été développés pour répondre aux problématiques exposées et à divers besoins en calcul et ou en optimisation de ces calculs. Nous abordons ici trois applications dans le contexte scientifique de la simulation de muons par la méthode de Monte Carlo. La première application, une simulation de Monte Carlo muonique avait été développée avant le début de la thèse, et nous présenterons uniquement les divers travaux ayant permis son optimisation. Ensuite, nous détaillerons plus particulièrement deux applications en raison de leur caractère innovant et des résultats qu'elles fournissent. Le premier développement innovant concerne un outil de profilage d'application parallèles écrites en C++ a été développé grâce au paradigme de programmation orienté aspect.

2 – Optimisation d'une simulation de Monte Carlo : `tomusim`

2.1 – Modifications apportées

2.1.1 – Passage en C++

Les défauts présentés dans le chapitre précédent nous ont amené à revoir en profondeur l'analyse et la programmation de `tomusim` (Schweitzer et al., 2013). Une des premières modifications majeures a été d'abandonner Python. En effet, `ROOT` étant développé en C++, il expose naturellement une interface de développement C++ et ne nécessite pas d'encapsulation, comme ce qui était fait avec `PyROOT`. Ceci a permis de réduire le nombre de composants impliqués. Une fois cette étape réalisée (quasiment de façon transparente avec l'application d'origine, le mimétisme allant jusqu'aux noms de variables), un profilage a été réalisé, ce qui a mis en évidence le fait que beaucoup de temps était perdu dans l'application à convertir des chaînes de caractères en nombres à virgule flottante (les positions et les charges électriques des muons), et réciproquement, afin de les fournir à MMC en vue de la simulation de la propagation ou bien de les récupérer en provenance de MMC, une fois cette simulation effectuée.

2.1.2 – Utilisation de la JNI

Par ailleurs, ce changement a permis de modifier l'interfaçage entre `tomusim` et MMC. Java dispose de la Java Native Interface (JNI) (Gordon, 1998). Cette interface permet d'exécuter du code C++ dans du Java et, inversement, de manipuler des objets Java dans un code C++. Ainsi, au lieu d'utiliser les entrées et les sorties standards, il suffit de manipuler directement, dans le code en C++, un objet Java, en appelant ses méthodes avec les bons arguments. Cette technique a permis de gagner beaucoup de temps étant donné qu'on ne passe plus par des conversions de nombres à virgule flottante vers des chaînes de caractères, et inversement, pour communiquer les informations sur les particules à propager ou récupérer les informations concernant la propagation de ces particules. Néanmoins, le développement avec la JNI est un peu plus complexe, car, même si on ne convertit plus des nombres à virgule flottante en chaînes de caractères, le passage du tableau qui contient les différents arguments nécessite, quant à lui, une conversion (pour s'assurer que les données soient correctement représentées). Toutefois, l'impact reste minime. L'utilisation de l'interface reste également beaucoup plus verbeuse que du Java « natif » et bien moins intuitive : par exemple, l'objet Java n'est pas représenté en tant que classe C++, mais différentes méthodes de la JNI doivent être appelées pour exécuter une méthode du Java.

À noter qu'il a fallu quelques changements dans la classe Java utilisée dans MMC pour la propagation. Puisqu'à l'origine, les échanges se faisaient *via* les entrées-sorties standards, toute l'initialisation de la MMC était implémentée dans la méthode `main()`. Il a donc fallu rendre la classe `Tomuvol` encore plus modulaire pour pouvoir l'utiliser réellement en tant qu'objet, sans passer par le point d'entrée.

2.1.3 – Gestion propre des flux stochastiques

Afin d'assurer l'indépendance des réplications, il fallait revoir la génération des flux stochastiques. Dans un premier temps, le plus urgent était d'abandonner l'usage du LCG dans MMC. Il existe des implémentations de MT pour Java, et c'est vers ce générateur que nous nous sommes tournés.

`tomusim` et MMC devaient partager le même statut (qui caractérise l'état du générateur, et par conséquent détermine le prochain nombre pseudo-aléatoire généré). Ainsi, en plus de s'échanger les informations sur la particule, MMC et `tomusim` s'échangent maintenant le statut du générateur

MT afin de pouvoir l'utiliser en toute continuité (pendant que MMC tire des nombres, `tomusim` ne fait qu'attendre le retour du statut).

Egalement, le *random spacing* devait être abandonné. Pour que l'utilisateur puisse distribuer la simulation, il doit pouvoir disposer de flux stochastiques dont l'indépendance est garantie. Pour ce faire, nous avons utilisé *Dynamic Creator* (DC) de MT (Matsumoto & Nishimura, 1998a). Celui-ci permet de paramétrer le générateur MT et de construire autant de générateurs que nécessaires, l'indépendance entre ces générateurs étant garantie par les concepteurs de DC. Grâce à DC, nous avons donc généré 499 MT différents (pour en avoir 500, avec celui d'origine). Nous avons vérifié la qualité statistique des 500 MT différents grâce au jeu de tests « *Big Crush* » de la bibliothèque TestU01 (L'Ecuyer & Simard, 2007). Mais ceci n'était évidemment pas suffisant pour lancer des simulations à grande échelle. Nous avons donc ensuite appliqué, sur chaque MT généré, la méthode du « *Sequence Splitting* ». Cette méthode consiste à découper la séquence de nombres pseudo-aléatoires associée à un générateur en plusieurs sous-séquences espacées régulièrement. Grâce aux propriétés devant être vérifiées par les générateurs de nombres pseudo-aléatoires, on s'assure que chaque sous séquence est indépendante des autres. De plus, en raison des propriétés mathématiques du MT, il est facilement possible de générer des sous-séquences grâce à un programme de « *jump-ahead* » (Haramoto et al., 2008) qui permet de "sauter" d'une position à une autre dans la séquence, et ceci en un temps constant, quelle que soit la longueur du saut. Ce type de programme permet d'éviter de dérouler la séquence en générant les nombres un par un, et nous avons ainsi pu rapidement générer 1 000 séquences par générateur.

L'utilisation conjointe du paramétrage de MT (avec DC) et de la technique de *Sequence splitting* (grâce à l'algorithme de *jump-ahead* cité ci-dessus) nous a permis d'offrir au manipulateur jusqu'à 500 000 séquences indépendantes de nombres pseudo-aléatoires. Les statuts de ces 500 000 séquences ont été stockés dans des fichiers binaires. Il y a un fichier par générateur créé par DC, donc au total 500 fichiers contenant chacun 1 000 statuts. Ainsi, l'utilisateur final n'a plus qu'à lancer `tomusim` avec un indice de séquence, ID, utilisé de la sorte :

```
MT_ID = ID mod 500;

Seq_ID = ID / 500.
```

Ce mécanisme permet de privilégier la sélection de générateurs différents avant de changer de séquence. Au lancement de `tomusim`, la valeur de « ID » permet de choisir (avec l'indice `MT_ID`) un fichier de statuts, correspondant à un générateur MT, puis de lire dans ce fichier le statut à utiliser (avec l'indice `Seq_ID`). Les paramètres du MT sont alors transmis à MMC pour que

tomusim et MMC utilisent le même générateur de nombres pseudo-aléatoires sur les 500 disponibles.

Plus tard, nous nous sommes aperçus des limites de l'utilisation d'un unique flux stochastique pour tomusim et MMC. En effet, lors du développement, et même après, pour pouvoir mesurer l'impact d'un changement de code, ou d'un changement de milieu de propagation, il est préférable de pouvoir comparer des flux de particules identiques. Si l'on n'utilise qu'un seul flux pour la génération et pour la propagation des particules, on se retrouvera alors inévitablement avec des particules différentes en cas de changement. C'est ainsi que nous avons séparé les flux stochastiques de tomusim et MMC : une séquence d'un générateur MT pour tomusim (génération des particules) et une autre séquence, d'un autre générateur MT (le suivant) pour MMC (propagation des particules). De la sorte, on se retrouve avec deux flux stochastiques dont l'indépendance est garantie par DC : un pour la génération et un pour la propagation. Ainsi, si on modifie le volume dans lequel les particules se propagent, on a toujours les mêmes particules.

2.1.4 – Modifications algorithmiques

Afin d'avoir des résultats plus précis, ou plus détaillés, nous avons également modifié les algorithmes utilisés dans tomusim. Ceci permettait également de gagner en performances.

La première modification algorithmique importante a été la suppression du détecteur. En effet, initialement, dans tomusim, un détecteur, qui peut accepter ou non une particule qui lui arrive, était modélisé et simulé. Cependant, ce modèle de détecteur n'était pas réaliste et cela aboutissait à des résultats faux, tout en ajoutant par ailleurs un temps de calcul inutile (une particule rejetée par le détecteur suppose la génération d'une nouvelle particule à propager). Nous avons fait le choix de la simplification algorithmique suivante : une fois qu'une particule arrive au plan de détection (*i.e.*, dans le plan où se trouverait le détecteur s'il était simulé), on accepte la particule si elle a une énergie non nulle. L'acceptation de la particule au niveau du détecteur sera gérée ultérieurement, avec une autre simulation de Monte Carlo, spécifique au détecteur.

Une autre modification a été effectuée dans le modèle utilisé par MMC pour la propagation des muons. MMC utilise deux méthodes pour propager les muons (Chirkin & Rhode, 2004) : soit une simulation de Monte Carlo, soit une méthode analytique. Le choix entre les deux méthodes est fait, soit en fonction d'une énergie de coupure e_{cut}, soit en fonction d'une énergie de coupure relative v_{cut}. Ce seuillage permet de définir la précision de la propagation. Quand

l'énergie d'une particule est inférieure à l'énergie de coupure, sa propagation n'est alors plus simulée, mais estimée *via* des calculs (en moyenne). De même, si la perte d'énergie d'une particule pour un pas de propagation est inférieure à la l'énergie de coupure relative, alors le modèle analytique sera utilisé pour ce pas. Si cette approche s'appuyant sur un modèle analytique est plus rapide, elle est surtout bien moins précise puisqu'elle fait appel à des moyennes. Elle n'est donc utilisée que sur les particules de basse énergie, dont la probabilité qu'elles traversent la cible est faible, ou sur des pertes d'énergies minimes par rapport à l'énergie initiale de la particule où l'impact de l'utilisation de moyennes sera négligeable. Néanmoins, l'énergie de coupure et l'énergie de coupure relative étaient 100 fois trop hautes, par rapport aux besoins de tomusim, ce qui conduisait à des résultats erronés en basse énergie. Il a fallu réajuster ces valeurs d'énergie de coupure, malgré le coût ajouté au temps de simulation.

Une autre modification pour accélérer la simulation a été de revoir la surface de génération des muons. En effet, avant de simuler la propagation des muons, il est important de les générer de telle sorte que la cible se retrouve entre la surface où ils seront générés et le détecteur. À l'origine, tomusim utilisait une surface de génération carrée, de 20 mètres carrés, située à 4 km du détecteur. Le défaut d'une telle surface est qu'elle peut potentiellement être trop grande, et générer des muons qui passeront à côté de la cible, voire à côté du détecteur, ce qui est donc du temps de simulation perdu. Nous avons donc revu la surface de simulation pour qu'elle maximise les chances pour que les muons arrivent vers le détecteur et la cible. Cette surface de génération est devenue dépendante de l'énergie initiale du muon E. Cette dernière ainsi que la direction du muon θ, en coordonnées sphériques, sont donc générées avant sa position initiale. On se retrouve donc avec un flux, Φ, dépendant de E et θ, ainsi qu'avec une surface S dépendante de E (alors qu'auparavant S était constante). Outre le fait d'améliorer le temps de calcul de la simulation en évitant des propagations non pertinentes, cela a entraîné d'autres modifications, d'ordre mathématique, nécessaires pour conserver la cohérence des calculs. Lors de la simulation, l'*exposure* (exprimée en s^{-1}) est calculée. Il s'agit d'une grandeur qui permet de mesurer le nombre de muons attendus par seconde dans cette configuration. Étant

données l'énergie E et la direction (θ,φ), l'*exposure* est calculée en intégrant le flux sur l'énergie, puis sur la surface de génération dépendant de cette énergie : $\int_E \Phi(E,\theta)\times S(E)\ dE$. Alors qu'auparavant, il était possible de sortir S de l'intégrale, il n'est ici plus possible de le faire, étant donné que S dépend également de E. Il est à noter que le temps simulé s'obtient en divisant le nombre d'évènements (nombre de particules générées) par l'*exposure*.

Une dernière modification (la plus importante) a permis d'augmenter de façon significative la précision de la simulation. Jusqu'alors, pour la propagation des particules, MMC utilisait un pas de taille de fixe, de 10 mètres. C'est-à-dire qu'il effectuait la propagation de la particule et l'application des effets physiques tous les 10 mètres. Ceci implique que les changements de milieux (ici caractérisés par leur densité) ne sont pas finement détectés sur une cible de la taille du Puy-de-Dôme, dont la distance de parcours est de moins de 5 km. L'algorithme qui gère la propagation des particules a alors été amélioré pour prendre en compte les changements de densité, et les positions de ces changements (les interfaces). À chaque itération de la propagation de la particule, les interactions physiques sont simulées. Plus le pas de propagation est grand, moins précise est la simulation. Par ailleurs, plus le pas est important, moins la détection des interfaces est fine. Il a donc fallu implémenter un pas de *tracking* adaptatif. Néanmoins, le changement de milieu ne peut être détecté qu'à l'issue de la propagation de la particule sur les 10 mètres, lorsque l'on compare la densité initiale avec la densité finale. Il a donc fallu implémenter un mécanisme de roll-back de la simulation, qui permet à la simulation de revenir à un état précédemment sauvegardé. Ici, le principe est qu'en cas de détection d'un changement de milieu sur les 10 mètres, un *roll-back* de la simulation est effectué à son état avant le début de la propagation de la particule sur ces 10 mètres, et le pas de propagation est réduit de moitié pour les propagations suivantes jusqu'à ce que l'interface ne soit plus détectée. À ce moment, on effectue encore un *roll-back* en réduisant le pas de moitié. Et ainsi de suite, jusqu'à ce que l'interface soit détectée à 30 cm près. Après le passage de l'interface, la propagation continue alors normalement avec un pas de 10 mètres, jusqu'à l'interface suivante. L'implémentation de ce mécanisme sera expliquée en détails

dans la section suivante dans l'ensemble de l'explication de l'algorithme de propagation de la particule.

L'impact du *tracking* adaptatif sur le temps de simulation est extrêmement important. Une solution étudiée pour tenter de minimiser cet impact a été d'ajouter une approche approximative simple, avant la propagation de la particule. L'objectif de cette étape analytique, bien moins lourde que la simulation de Monte Carlo avec le *tracking* adaptif, est d'évaluer quelle quantité de matière la particule traverserait durant la simulation et de vérifier si elle aurait assez d'énergie pour atteindre le détecteur. Il suffit donc faire une propagation très rapide sans prendre en compte les processus pour estimer a priori quelle quantité de matière et avec quelle densité moyenne la particule va traverser. De même, on calcule le grammage de la particule qui correspond à la « portée » théorique de la particule pour une densité (moyenne) donnée, *i.e.* la quantité de matière traversée sur le chemin. Si une particule a une quantité de matière à traverser qui est supérieure à son grammage, elle n'est même pas simulée. En effet, l'estimation permet de savoir que la particule devra traverser une distance supérieure à sa portée théorique : jamais elle n'arrivera au détecteur. Néanmoins, devant les limites de tomusim, cette solution n'a pas eu le temps d'être éprouvée, une fois son implémentation terminée, le logiciel ayant été abandonné avant.

2.2 – Performances au fil des évolutions

Le premier logiciel `tomusim` écrit en Python prenait 16 heures pour accepter 10 000 particules, avec pour énergie initiale une énergie comprise entre 1 GeV et 50 TeV. Le processeur utilisé était alors un Intel Xeon X5650 (cadencé à 2,67 GHz, performance maximum à 3,06 GHz). Un détecteur équivalent à celui simulé dans `tomusim` aurait pris la même quantité de données en 15 heures.

2.2.1 – Gains séquentiels

À l'issue de la réécriture en C++, et avec quelques optimisations, notamment pour supprimer les appels aux fonctions génériques de conversions entre nombres à virgule flottante et chaînes de caractères pour la communication avec MMC (elles ont été remplacées par des fonctions spécialisées), de premiers gains de performances ont été obtenus. Dans le même cadre expérimental, donc sur le même processeur que précédemment, la simulation prenait alors 9,5 heures. On avait alors un gain de performance de 1,73X par rapport à la simulation initiale.

Pour la deuxième étape, consistant à s'affranchir de la communication entre `tomusim` et MMC et à optimiser le code en utilisant la JNI, le gain de performances a été bien plus important. La simulation, toujours dans les mêmes conditions, tournait alors en approximativement 1 heure, soit, par rapport à la simulation initiale, un gain de performance de 16,22X.

2.2.2 – Distribution

Dès lors, la simulation a été distribuée, et les premières optimisations ont été ajoutées, comme la suppression du détecteur de la simulation (celui-ci sera géré par une autre simulation dédiée) et l'optimisation de la surface de génération. Les tests suivants furent réalisés sur une autre machine, équipée d'un processeur Intel Xeon E5-4620 (32 cœurs physiques, 64 cœurs logiques), cadencé à 2,20 GHz avec une performance crête à 2,60 GHz. La simulation a été exécutée sur un nombre de particules à l'échelle des statistiques attendues par les physiciens par rapport à une observation réelle avec le détecteur, le but étant d'avoir en sortie 10 millions de particules acceptées (afin d'avoir des données à comparer). Nous avons également mesuré deux indices : le temps de simulation, mais également la quantité de nombres pseudo-aléatoires tirés par instance distribuée. Les résultats se trouvent dans le tableau 1, pour la comparaison des temps de calcul et les gains de performance associés. La figure 1 présente, quant à elle, la comparaison entre les temps de calcul et la quantité de nombres pseudo-aléatoires tirés par instance.

Tableau 4-1 : Comparaison des temps de calcul pour la simulation distribuée

Distribution	1	2	4	8	16	32
Temps de calcul	60h 9m 32s	29h 54m 45s	15h 26m 12s	8h 3m 34s	3h 48m 44s	2h 20m 48s
Speed-up	1X	2,01X	3,90X	7,46X	15,78X	25,64X

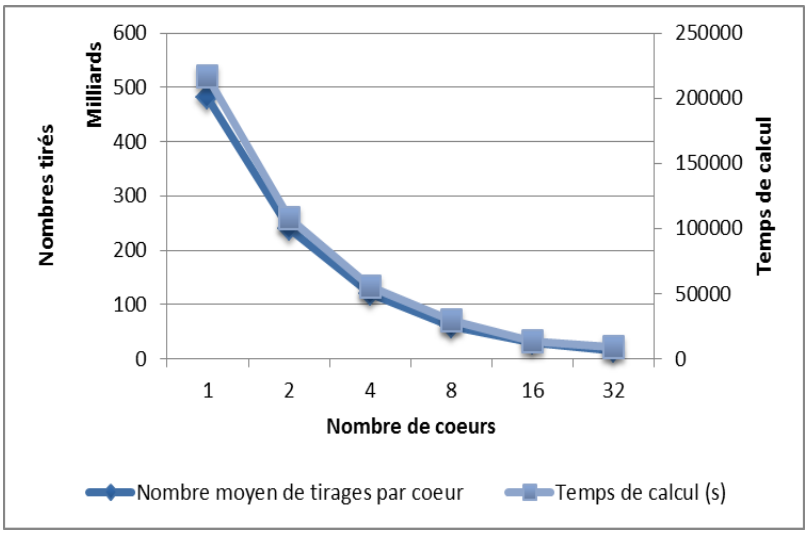

Figure 4-1 : Distribution de la simulation sur plusieurs cœurs

La figure 1 met bien en évidence le fait que la quantité de nombres pseudo-aléatoires tirés est globalement proportionnelle au nombre d'évènements demandés, les petites fluctuations venant justement du fait que la simulation est une simulation de Monte Carlo, donc, non-déterministe.

En revanche, on peut apercevoir sur le tableau 1 que le gain de performance n'est pas tout à fait celui attendu. On s'attendrait à avoir un gain de performance de 32X par exemple pour la distribution sur 32 cœurs physiques. Néanmoins, on peut constater que plus la simulation est distribuée plus le gain de performance tombe. D'après la loi d'Amdahl, on peut effectivement se heurter à un plafonnement des performances inférieur à la capacité maximale théorique en fonction du pourcentage de calculs qui resteront séquentiels. Pour tenter d'y apporter une explication plus claire, nous avons mesuré la proportion de temps de simulation qui peut être réellement distribué. En effet, quelle que soit la simulation, les temps d'initialisation et de clôture sont incompressibles. Seules la génération et la propagation des particules peuvent être distribuées. Les résultats sont présentés dans la figure 2.

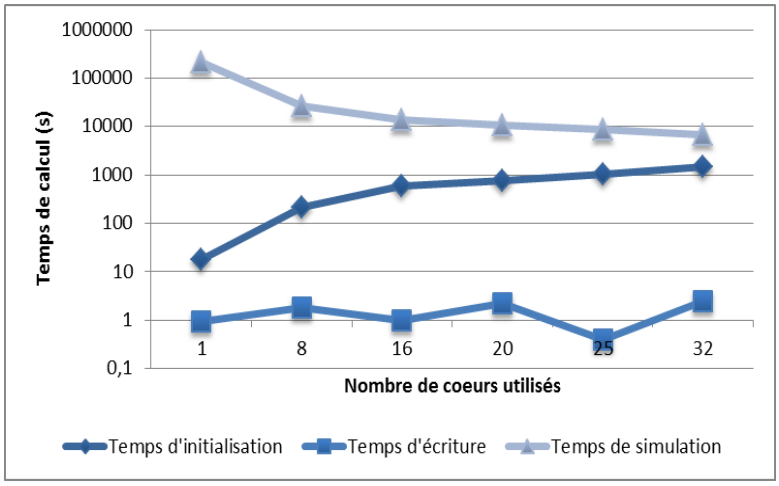

Figure 4-2 : Temps passé dans les différentes parties de `tomusim` distribué

Sur la figure 2, on peut constater un phénomène qui explique l'absence de gain de performance linéaire comme attendu. L'initialisation prend du temps, et plus on distribue la simulation, plus elle en prend (alors que le temps de calcul pur lui-même diminue). Une partie du problème a été réglé en désynchronisant le lancement des simulations, pour éviter qu'elles ne lisent toutes en même temps le fichier de données. En effet, le disque était partagé et incapable de suivre le débit imposé par les simulations qui se ralentissaient donc toutes entre elles.

Finalement, par rapport à la simulation initiale qui demandait environ 16 heures pour 10000 particules acceptées, les optimisations de la version séquentielle et la distribution sur 32 cœurs physiques apportent conjointement un gain de performance de l'ordre de 400X (16,22X × 25,64X) et donc des résultats en un peu plus de 100 secondes sur un seul nœud de calcul.

2.2.3 – Support du tracking *adaptatif*

Forts de ces gains de performances, nous avons alors implémenté le *tracking* adaptatif. Malheureusement, en raison de la lourdeur de la simulation à faire, les temps de calculs ont alors « explosés ». Si nous avions voulu exécuter notre simulation distribuée avec le *tracking* adaptatif dans les mêmes conditions, même en réduisant le spectre des basses énergies à 5 GeV (l'expérience montrant qu'en dessous de ces niveaux d'énergie, les particules ne passent pas), la simulation aurait eu besoin d'au minimum 1 an de calcul, ce qui n'est pas du tout envisageable.

Il aurait été possible de rendre la simulation bien plus performante en modifiant l'algorithme de façon à pouvoir anticiper les changements de densité en extrapolant la position d'arrivée de la particule sans simuler sa propagation. Il serait ainsi possible d'évaluer l'éventuelle différence de densité entre les positions de la particule avant et après la propagation. Cette idée n'a néanmoins pas été implémentée en raison du temps de travail que cela aurait représenté par rapport aux alternatives disponibles.

2.2.4 – Limites

La simulation souffrait d'autres problèmes que ceux déjà présentés : en raison du mélange de Java (*via* la JNI) et C++, il n'était plus possible de profiler la simulation pour tenter de trouver les causes des problèmes. Les profileurs étaient inopérants : `valgrind` cessait de fonctionner, ou n'arrivait pas à lancer la simulation ; `gprof` donnait des résultats difficilement interprétables ; quant aux profileurs en Java, sur la plate-forme de travail, ils ne pouvaient pas s'attacher à des objets utilisés *via* la JNI, quand ils n'interrompaient pas tout simplement, de manière abrupte, le fonctionnement de la machine virtuelle Java. La simulation était devenue extrêmement instable en raison de la trop grande quantité de mémoire utilisée. Pour pouvoir fonctionner, elle utilisait un modèle de 1 Go de données binaires qui correspondent à un histogramme en 3D (au format de `jHEPWork`) des densités du Puy-de-Dôme (uniformes pour l'instant). Ce modèle était donc chargé en mémoire par la JNI, en plus des données nécessaires pour effectuer le *roll-back* de la simulation, ce qui posait des problèmes de gestion de la mémoire pour la machine virtuelle Java.

Ces dépendances (fichier de données lourd, MMC, JNI, Java, `ROOT`) rendaient également le déploiement de la simulation particulièrement difficile. Il n'était pas envisageable de déployer la simulation sur la grille, et encore moins sur des accélérateurs matériels tels que les Xeon Phi ou les *General Purpose Graphical Processing Units* (GPGPU).

Enfin, l'existence de cadriciels éprouvés de simulation en Physique des Particules tels que `Geant4` (Agostinelli et al., 2003) ont rendu `tomusim` totalement caduc et ont abouti à son abandon définitif au profit d'une nouvelle simulation écrite à l'aide de `Geant4`, `tmvg4sim`.

3 – Profilage d'applications C++ : `acprof`

3.1 – Motivations

La problématique du profilage d'applications C++ s'est posée très tôt, avec `tomusim`. Les premières optimisations sur `tomusim` ont été réalisées grâce à un profilage réalisé par `valgrind`. Comme expliqué dans la précédente section, ceci a conduit à reprendre la façon dont `tomusim` communiquait avec MMC (Muon Monte Carlo). Au lieu de lancer une instance séparée (c'est-à-dire un nouveau processus) de MMC et de communiquer avec elle *via* les entrées et les sorties standard, ce qui se révélait coûteux en temps CPU, `tomusim` utilisait la JNI (Java Native Interface) pour communiquer avec MMC. La JNI permet de manipuler directement dans le programme C++ les objets Java. Ce qui réduit la complexité de développement, tout en réduisant également l'*overhead* lié aux conversions multiples qui étaient faites précédemment. Cela implique également que pour fonctionner la JNI lance discrètement une machine virtuelle Java dans un thread du processus hôte, `tomusim`.

D'un point de vue du profilage, cela a des conséquences notables. Généralement, les profileurs ne suivent pas les processus fils, sauf si on leur demande explicitement. Dans le cadre de `valgrind` et de la première version de `tomusim`, avec ce fonctionnement, on se retrouvait donc à ne profiler que `tomusim`, étant donné que la machine virtuelle Java (JVM) était dans un processus fils séparé. En revanche, pour `tomusim` avec la JNI, la machine virtuelle étant dans un nouveau thread, `valgrind` va donc tenter de profiler la JVM. Il est important de noter qu'il profile la JVM et non MMC. `valgrind` n'a pas connaissance de ce qu'exécute la JVM, que cela correspond à un programme Java. Tout ce que `valgrind` voit ce sont les appels systèmes de la JVM, ainsi que les appels à ses fonctions internes. C'est donc totalement agnostique des classes Java. Dans ce cas donc, on se retrouve à profiler `tomusim` et la JVM. Ce qui pose techniquement deux problèmes : on n'a aucun intérêt à profiler la JVM, cela ne fera que brouiller le profil final. Le second problème est plus complexe...

En effet, afin de pouvoir profiler le plus précisément possible une application, `valgrind` émule une machine virtuelle dans laquelle il va faire tourner une application invitée afin de pouvoir l'observer. Cela signifie notamment qu'il émule le processeur et ses instructions, ainsi que la mémoire. Cela permet de constater quels appels système l'application fera, quels appels interne elle fera, mais également qu'elles seront les instructions processeurs utilisées ainsi que la mémoire consommée, et comment. A partir de ces données, `valgrind` peut notamment estimer le coût processeur théorique de l'exécution d'une application grâce au coût théorique en « *clock ticks* » des instructions du processeur. Mais l'envers de ce mécanisme est que `valgrind` doit impérativement

connaître les instructions du processeur afin de pouvoir les émuler et évaluer leur performance. A titre d'exemple, les jeux d'instructions vectoriels récents tels qu'AVX ou AVX2 ne sont pas complètement supportés par `valgrind`. De plus, avec les nouvelles versions de `valgrind`, la quantité de jeux d'instructions supportés augmente. Cela signifie que plus une version de `valgrind` est ancienne, moins elle supporte de jeux d'instructions récents.

D'autre part, la JVM est compilée avec du code dynamique, qui permet de sélectionner les meilleurs jeux d'instructions possibles pour l'exécution de la JVM. C'est-à-dire qu'à l'exécution de la JVM, elle va sélectionner les jeux d'instructions supportés par le processeur pour exploiter les plus récents et les plus performants. L'objectif est d'offrir le maximum de performance possible *via* les instructions les plus efficaces du processeur, sans nuire à la portabilité de l'application.

Cela signifie donc qu'on peut se retrouver avec une inadéquation entre les usages de la JVM et les capacités de `valgrind`. Sur nos machines de calcul, les instructions SSE4.2 et AVX(2) étaient disponibles et la JVM tentait donc d'en tirer profit. Néanmoins, les systèmes d'exploitation utilisés, Red Hat Enterprise Linux 5 et 6, venaient avec une version trop ancienne de `valgrind` qui ne supportait pas ces jeux d'instructions. A noter qu'à l'heure de rédaction AVX2 et AVX ne sont toujours pas complètement supportés. On se retrouvait donc dans le cas de figure où l'outil de profilage était totalement incapable de profiler l'application : c'était le second problème technique que l'on a rencontré. D'autant qu'il est possible de demander à la JVM de ne pas exploiter ces instructions vectorielles, mais dans ce cas, la JVM crashait à son initialisation dans `tomusim`. Il est possible que cela soit lié aux contraintes (fortes) imposées par `valgrind` sur la mémoire, et notamment sur la pile applicative. `valgrind` n'était donc plus une option pour profiler `tomusim`.

Comme indiqué dans l'état de l'art, d'autres outils de profilage existent, et ils auraient pu se substituer à `valgrind`, néanmoins le premier problème subsistait : cela ne présentait aucun intérêt de profiler la JVM. A ce problème se rajoutaient d'autres problèmes : complexité de mise en œuvre, limites du profilage, coût de la licence, etc. Nous en sommes donc arrivés à la conclusion que développer un profileur simple pourrait régler le problème. Etant donné qu'il existe déjà de nombreux profileurs pour les applications s'appuyant sur OpenMP, MPI, etc., nous avons fait le choix technique de ne pas les supporter, en limitant notre support aux threads POSIX, de la bibliothèque `pthread`.

3.2 – Implémentation

Pour notre profileur, nous avons utilisé une approche inédite en C/C++, mais néanmoins courante en Java : l'utilisation de la programmation orientée aspect (AOP – *Aspect Oriented Programming*), initialement formalisée dans (Kiczales et al., 1997). On peut trouver `AspectJ` en Java (Pearce et al., 2007) qui permet de développer grâce à l'AOP en Java. Grâce à cette implémentation de l'AOP, on trouve des profileurs tels que celui décrit dans (Pearce et al., 2007) qui permettent le profilage d'applications Java.

C'est cette approche que nous avons voulu transposer au C++ afin de pouvoir profiler `tomusim`. Deux implémentations de l'approche aspect existent en C++ : `FeatureC++` (Apel et al., 2005) et `AspectC++` (Spinczyk et al., 2002). Les deux implémentations ne sont pas équivalentes et ne fournissent pas les mêmes fonctionnalités. `FeatureC++` permet de faire des implémentations partielles de classes qui seront par la suite spécialisées selon le besoin. Il fournit également le mot clé « super » (tiré du Java) qui permet d'appeler des éléments de la classe mère d'une classe. Enfin, il permet d'étendre à la volée des classes qui existent déjà. Il ne propose donc pas les fonctionnalités recherchées. En revanche, `AspectC++` vient avec les fonctionnalités cherchées pour pouvoir faire du profilage d'application, à savoir pouvoir rajouter du code de façon transversale, autour d'une base de code déjà existante et indépendante.

Le paradigme de programmation orientée aspect permet d'avoir une approche du développement transverse par rapport aux paradigmes standards. Alors que la conception traditionnelle d'un programme est « verticale » avec un enchaînement de fonctions et d'appels de ces fonctions, la programmation orientée aspect vient rajouter un niveau de modularité transverse au programme. Il est alors possible de définir des fonctions, des modules, qui seront greffées au programme à des endroits précis. Ces fonctions peuvent dès lors changer le comportement du programme, ou en tout cas, en extraire plus d'informations, voire valider le comportement. On peut par exemple imaginer deux modules, l'un de « release » l'autre de « *debug* ». Le premier permettrait de faire des tests extrêmement légers sur les adresses utilisées par le programme (typiquement, juste vérifier qu'il n'y a pas de pointeur `null`), tandis que le second de débogage pourrait aller plus loin et vérifier que les adresses sont bien dans l'espace d'adresse du programme. Ceci permet d'illustrer qu'on a toujours le même programme, dont on n'a pas changé le code. Le simple fait de changer les aspects utilisés permet en revanche de moduler son comportement.

Ce sont les fonctionnalités qu'offre `AspectC++`. Grâce à `AspectC++`, il est possible de définir des « *pointcuts* », des points de coupure. Ce sont à ces points de coupure qu'`AspectC++` pourra rajouter le code de l' « *advice* », qui est la véritable implémentation de l'aspect, celle qui pourra éventuellement modifier le comportement de l'application, comme décrit précédemment. L'étape durant laquelle l'*advice* est inséré à son point de coupure est connue sous le nom de « *weaving* ». Dans `AspectC++`, la finesse des points de coupure est de l'ordre de la fonction. C'est-à-dire qu'il est possible d'insérer un *advice* en début, en fin de fonction ou autour d'une fonction. Le point de coupure autour (d'une fonction) n'est que la somme d'un début et d'une fin, mais cela permet de faciliter le développement, voire de garder des variables entre le début et la fin. Grâce à ces points de coupures, il est donc possible de profiler une application avec une finesse de l'ordre de la fonction. On ne pourra pas donner le coût à l'instruction, ni même à la ligne, mais au niveau de la fonction. Par ailleurs, on constate également, qu'il y a une étape de « *weaving* ». Contrairement au Java où c'est totalement transparent, cela implique qu'il y aura un changement dans la chaîne de compilation du programme pour prendre en compte l'aspect. Il faut en effet insérer les *advices* dans le code qui est sujet aux aspects aux points de coupure définis. On a donc une étape initiale de compilation où l'outil de *preprocessing* d'`AspectC++` va réaliser le « *weaving* » en analysant les fichiers d'aspect et le code source, pour générer un code source complet, qui contient l'ensemble. Ce code source complet sera alors compilé normalement par le compilateur d'origine, et l'application avec son aspect sera prête. Cela signifie notamment qu'il est également possible d'activer ou non l'aspect. Si on compile sans cette première étape, on se retrouve alors avec une application « nue ». Et c'est également à cette étape initiale qu'on choisira quels aspects activer et quels aspects ignorer.

Cette méthode de profilage va donc suivre l'exécution de l'application. A chaque entrée (et sortie) d'une fonction, des informations de performance seront collectées et traitées. Tant que l'application est dans la fonction et n'en appelle pas une autre, le profileur ne sera pas au courant de ce qui se déroule. Ce profilage qui suit l'exécution du programme est, selon la nomenclature établie dans (Malony et al., 2011), du profilage synchrone. Par opposition, les auteurs définissent le profilage asynchrone comme une méthode où l'application profilée est régulièrement observée (par exemple, sur la base du temps) pour vérifier dans quel état elle est, dans quelle fonction elle est. Cette seconde méthode de profilage présente un avantage non négligeable d'être beaucoup plus fine dans ses analyses, puisqu'elle n'est pas tributaire de l'architecture de l'application pour récupérer des données. Néanmoins, son défaut est qu'elle rajoute un *overhead* plus important, en raison des interruptions fréquentes de l'application réalisées afin de récupérer les données de

performances. La documentation de `valgrind` (“Callgrind: a call-graph generating cache and branch prediction profiler,” 2013), par exemple, explique que `valgrind` a un *overhead* minimum de l’ordre de 0.25X ; l’application est donc quatre fois plus lente, au minimum, dès qu’elle tourne au sein de `valgrind`. Dans notre cas, faire de l’asynchrone avec une conception par l’orienté aspect n’aurait pas fait grand sens également.

Avec `AspectC++`, un aspect n’est, ni plus ni moins qu’une classe C++ (dénommée « aspect » ici) dans laquelle on regroupe les différents *advices* ainsi que les points de coupure sur lesquels appliquer les *advices*. Ainsi, lors qu’`AspectC++` fera l’étape de *weaving*, il mettra l’aspect dans une classe. Par ailleurs, les points de coupures qu’il supporte font qu’`AspectC++` va se contenter de faire une diversion autour de la fonction de base qui aurait dû être implémentée. Dans cette diversion, il va insérer le code de l’aspect (en appelant la bonne fonction de la classe) puis réellement appeler la fonction, qu’il aura renommée avec un nom interne, puis à nouveau appeler le code de l’aspect. C’est ainsi qu’il est possible d’implémenter un aspect au début, à la fin ou autour d’une fonction. Bien évidemment, `AspectC++` va plus loin, en fournissant des mots clés et des fonctions supplémentaires qui permettent de manipuler le nom de la fonction, ses paramètres, etc. Néanmoins, l’avantage d’avoir une classe pour l’aspect signifie que l’on peut pousser les implémentations très loin, comme avoir des fonctions et des variables privées qui ne pourront être utilisées que par des *advices*, ou encore de définir un destructeur pour l’aspect afin de libérer des ressources ou pour avoir un comportement spécifique à la fin de l’exécution du programme.
Ainsi, pour implémenter le profileur avec `AspectC++`, il nous a fallu définir un aspect, que nous avons appelé `TProfiler`. Nous avons suivi le style de développement Taligent (Press, 1994). Nous avons ensuite défini deux points de coupure, comme montré sur le Code 1.

```
pointcut functions() = "% ...::%(...)";
pointcut profiler() = "% TProfiler::%(...)";
```

Code 4-1 : Points de coupure pour le profileur

Le premier point de coupure signifie qu’il va correspondre à toutes les fonctions, de toutes les classes, quels que soient leurs paramètres. Le second point de coupure sert lui à répondre à une problématique fréquente lorsque l’on développe un profileur. Il correspond à toutes les fonctions qui sont dans la classe `TProfiler`, peu importe leurs paramètres. Ce point de coupure est nécessaire pour pouvoir exclure les fonctions du profileur de son propre profilage. Autrement, on pourrait se retrouver à profiler le profileur. Ce qui n’est pas le but de l’exercice. Ainsi, pour définir nos *advices*, on le fera comme dans le Code 2.

```
advice execution(functions() && !profiler()) :before() { ... }
advice execution(functions() && !profiler()) :after() { ... }
```

Code 4-2 : Déclaration des *advices* pour le profilage

Sur le code 2, on peut constater plusieurs éléments importants. Tout d'abord nous définissons deux *advices*. Ils servent, comme expliqué précédemment, à récupérer des données de performance avant et après l'exécution de la fonction. Par ailleurs, on note que l'*advice* est utilisé lors de l'exécution de la fonction. En effet, `AspectC++` permet de choisir la finesse avec laquelle on peut vouloir utiliser l'*advice*. On peut choisir par exemple avant et après l'exécution de la fonction, ou avant et après l'appel de la fonction. Si cela peut sembler être identique, cela ne l'est pas, nous ne sommes pas au même niveau d'abstraction. Dans le cas de l'appel d'une fonction, le code sera inséré dans la fonction appelante et encadrera l'appel, tandis que dans le cas de l'exécution, le code sera inséré dans la fonction et encadrera le code initial. Dans notre cas, nous avons choisi l'exécution pour être au plus proche du code profilé. Enfin, on peut constater qu'il y a bien correspondance avec toutes les fonctions possibles sauf celles du profileur ; c'est une simple expression booléenne sur les points de coupure qui permet d'éviter de profiler le profileur.

Il a ensuite fallu se poser la question de quel coût mesurer, et comment. Dans un premier temps, l'objectif étant de prototyper le bon fonctionnement du profileur avec la programmation orientée aspect. Dans un souci de simplification et afin de nous concentrer sur l'implémentation plus que sur les compteurs de performances, nous nous sommes appuyés sur des compteurs simples fournis par Linux : compteur de tics d'horloge, ainsi que des compteurs d'utilisation du temps système et du temps utilisateur. Les deux derniers permettent de mesurer le temps que l'application passe dans son code propre ou dans les routines du système, ils sont obtenus grâce à l'appel système `getrusage()`, tandis que le premier compteur permet d'évaluer approximativement le nombre de tics d'horloge qu'il y a eu pendant l'exécution d'une fonction, que la fonction soit simplement en attente ou en train d'exécuter activement du code. Ce premier compteur est lui obtenu grâce à l'appel système `clock()`. Ce dernier appel permet également d'avoir une estimation du temps d'exécution total, puisqu'il suffit de le multiplier par l'intervalle de temps entre deux tics d'horloge. Le temps d'exécution total est important puisqu'il correspond à l'expérience qu'à l'utilisateur final quand il utilise l'application, `clock()` suffit amplement ici, puisqu'il ne sert qu'à valider une impression utilisateur et ne fournit pas une indication précise, contrairement à `getrusage()`. Linux fournit également d'autres compteurs de performance *via* son interface noyau « `perf_events` », mais dans un souci de simplicité et d'efficacité, nous avons fait le choix de ne pas les utiliser, notre souci du détail étant fixé sur l'implémentation du profileur afin qu'il fournisse

des résultats corrects et cohérents. De même, nous n'avons pas considéré la solution proposée par Intel *via* la bibliothèque `Pin` (Reddi, Settle, Connors, & Cohn, 2004) (Luk et al., 2005), celle-ci étant, en revanche, parfaitement appropriée pour une implémentation finale.

Dans l'*advice* avant l'exécution, le profileur doit réaliser deux tâches. La première est de replacer la fonction en cours d'appel dans son contexte d'exécution. L'objectif est de pouvoir maintenir en mémoire un arbre contextuel des appels. Cet arbre ressemble à ce qui est représenté sur la figure 3.

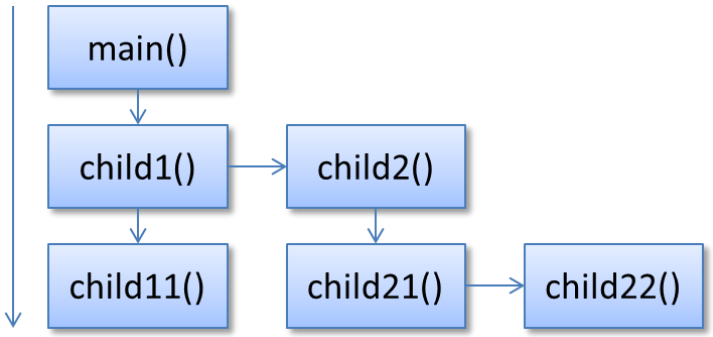

Figure 4-3 : Représentation de l'arbre contextuel des appels

On peut constater sur la figure 3 que l'arbre débute à la fonction `main()`. Ce comportement est attendu étant donné que la fonction `main()` représente le point d'entrée d'un programme. Ensuite, chaque nœud représente l'appel à une fonction dans un contexte particulier. C'est là qu'entre la notion de contexte. En effet, dans le cas de notre arbre, `child11()` et `child21()` peuvent être la même fonction dans le code. Néanmoins, comme le contexte d'appel n'est pas le même ; `child11()` est appelée par `child1()` et `child21()` par `child2()`, alors la fonction apparaît deux fois dans l'arbre. C'est la raison pour laquelle on parle d'arbre contextuel d'appel. La position du nœud dans l'arbre permet de situer le contexte d'appel. Ainsi, chaque nœud de l'arbre a un pointeur retour vers son parent (non représenté ici, pour ne pas alourdir le dessin), ce qui permet de remonter toute la pile d'appel. Par ailleurs, dans l'arbre, toutes les fonctions appelées par une fonction parente sont reliées entre-elles. En effet, la fonction parente n'a qu'un pointeur vers une des fonctions filles, et celles-ci sont rangées sous forme de liste chaînée.

Ainsi, dès qu'une fonction est appelée, il suffit de vérifier si elle a déjà été appelée pour ce parent. Si oui, il faut juste incrémenter le nombre d'appels et garder trace de l'état des compteurs de performance avant l'appel. Autrement, on se retrouve à ajouter un nouveau fils à la fonction parente, puis on initialise les compteurs de performance afin de pouvoir commencer le profilage. On

peut donc représenter un nœud de l'arbre avec la structure de données décrite dans le Code 3. Elle s'appuie sur la structure définie dans le Code 4.

```
struct TCallInfo {
    TCallInfo *   fParent;
    TCallInfo *   fNeighbor;
    std::string   fFunction;
    std::string   fFile;
    int           fLine;
    unsigned long fCalls;
    unsigned long fTotalTicks;
    TTimeInfo     fTotalTime;
    clock_t       fStartTicks;
    TTimeInfo     fStartTime;
    unsigned long fReferenceCount;
    TCallInfo *   fChildren;
};
```

Code 4-3 : Structure de données représentant un nœud de l'arbre contextuel d'appels

```
struct TTimeInfo {
    timeval fUserTime;
    timeval fKernelTime;
};
```

Code 4-4 : Structure de données qui stocke des données de compteurs de performances

On peut voir sur le Code 3 que chaque nœud contient un nombre important d'informations sur la l'appel de la fonction. En tout premier lieu, comme expliqué précédemment, on retrouve un pointeur vers la fonction parente, celle qui a réalisé l'appel à cette fonction. Le champ suivant, `fNeighbor`, lui permet d'établir la liste chainée des autres fonctions filles de la fonction parente. Les trois champs suivants permettent de caractériser la fonction : son prototype complet, le fichier où elle est implémentée, ainsi que le numéro de la première ligne où commence l'implémentation. Les champs suivants permettent de commencer à caractériser ses performances : `fCalls`, le nombre d'appels qu'il y a eu à cette fonction, étant donné ce chemin d'exécution, `fTotalTicks`, le nombre total de tic d'horloge qu'il y a eu sur les exécutions, ainsi que les différentes mesures de temps dès que la fonction était exécutée, grâce à la structure décrite dans le fragment Code 4, qui fait office de champ `fTotalTime`. On trouve ensuite les champs `fStartTicks` et `fStartTime` qui permettent de stocker l'état des compteurs au début de l'appel, afin de pouvoir établir combien

a été consommé durant l'exécution. On trouve alors un champ extrêmement important, `fReferenceCount`. Ce champ a été ajouté pour permettre de supporter les appels récursifs. Ainsi, plutôt que de dérouler un arbre sur une fonction récursive, ce qui peut se révéler très coûteux en temps de calcul ainsi qu'en mémoire, dès qu'on détecte que la fonction parente est la même que la fonction actuelle, on ne contente d'incrémenter le champ `fReferenceCount`, sans avancer dans l'arbre ni toucher les champs des compteurs de données de performance. La prise de données est toujours réalisée, toujours cohérente, mais sans risques. Enfin, le dernier champ, `fChildren` permet de pointer vers la première des fonctions filles de cette fonction, comme indiqué sur la figure 1.

A la fin de l'exécution d'une fonction, il suffit de commencer par décrémenter la valeur contenue dans `fReferenceCount`. Si elle est supérieure (strictement) à 0, alors nous sommes dans un appel récursif, et il n'y a rien de plus à faire. En revanche, dès que cette valeur atteint 0, il suffit de calculer les différentiels entre les données des compteurs et les données sauvegardées au début de l'appel de la fonction. Ce delta nous donne le temps consommé par la fonction. On notera ici que les données de performance sont donc inclusives : on a les données de performances de la fonction en elle-même, ainsi que les données de performances des fonctions filles incluses. On oppose ce modèle aux données exclusives, où l'on a uniquement les données de la fonction en elle-même sans la prise en compte des filles. Néanmoins, il est plus pertinent de considérer les données inclusives : une simple soustraction permet, si nécessaire, de se ramener à des données exclusives.

La fonction `main()` bénéficie également d'un traitement un peu particulier, étant elle-même une fonction particulière. C'est notamment elle qui reçoit la ligne de commande du programme, grâce à ses deux premiers arguments ; on ne considère pas ici le troisième qui ne contient que les données de l'environnement au démarrage de l'application. La question s'est posée de permettre de moduler le comportement du profileur, grâce à la ligne de commande du programme, pour éviter de devoir recompiler le programme à chaque fois que l'on veut changer la méthode de profilage. Un des objectifs était notamment de supporter deux méthodes de mise en forme des données de profilage. La méthode la plus simple étant un format de données textuel, où les données sont simplement affichées sur la console à la fin de l'exécution du programme. L'autre méthode étant de produire un fichier de sortie compatible avec ce que produirait `valgrind`, de sorte que les données de profilage d'`acprof` puissent être utilisées avec les mêmes logiciels, à commencer par `Kcachegrind` qui offre des mises en formes graphiques précieuses. Il était donc nécessaire que l'utilisateur final puisse choisir son format de sortie *via* la ligne de commande du programme. Ainsi,

nous avons défini un paramètre du type « `--acprof-out=callgrind` » qui permet de sélectionner le format de sortie compatible avec `valgrind`. L'absence de ce paramètre provoquant une sortie textuelle sur la console. Le véritable problème posé par cette méthode est qu'un tel argument ne doit jamais arriver au programme profilé, ne serait-ce que parce que si le programme vérifie scrupuleusement les paramètres qu'il reçoit, il retournera une erreur parce qu'il ne reconnait pas celui-ci. Il était donc nécessaire de rajouter un *advice* encapsulant la fonction `main()` afin de pouvoir intercepter les arguments passés au programme. En effet, `AspectC++` permet de manipuler les arguments reçus par une fonction, voire de les modifier. Dans notre cas, il suffit de *parser* (analyser) les arguments reçus par le programme. Si l'on repère que l'un des arguments est pour le profileur, alors on le prend en compte, et on ne le passe plus au programme. Pour ce faire, il faut réallouer la liste des paramètres du programme, et seulement remettre dans cette liste les paramètres qui ne sont pas ceux du profileur. Bien évidemment, une copie de l'ancienne liste des paramètres est conservée, afin de pouvoir la rendre au système une fois l'exécution de la fonction `main()` (et donc du programme) terminée. Avec cette méthode, le programme profilé n'a pas conscience de ne pas avoir la liste complète des arguments, ni du fait que certains ont été interceptés par le profileur.

3.3 – Premiers tests de validation

Afin de valider l'implémentation interne des structures de données, nous avons mis au point une première application de test, qui permettait de voir si l'arbre contextuel était bien créé, et s'il n'y avait pas de confusion entre différents appels. Cette application de tests a également été utilisée pour évaluer si les données de performance étaient proprement récupérées et conservées. Enfin, cette application de tests devait permettre de valider la bonne mise en forme des données pour l'utilisateur, quel que soit le format de sortie choisi.

Cette application a un fonctionnement très simple. Elle va réaliser des calculs mathématiques en fonction de l'entrée d'un utilisateur. Tout d'abord, elle va calculer le nombre de nombre premiers inférieurs à une certaine valeur, en utilisant le crible d'Ératosthène. Puis, elle va calculer des décimales de Pi, selon la précision donnée par l'utilisateur, en utilisant la formule de Madhava. Ces calculs sont d'abord faits simplement en appelant les fonctions qui les implémentent, puis, sont réalisés grâce à l'utilisation d'une classe C++ « Maths » qui encapsule l'appel vers les fonctions de base. Cette dernière méthode permet de vérifier la bonne prise en charge des classes C++, mais

également, elle rajoute un niveau dans l'arbre d'appel, et permet de vérifier que les appels sans la classe et ceux avec ne sont pas confondus dans l'arbre contextuel.

```
double Sqrt(double)
<- unsigned int FrequencyOfPrimes(unsigned int)
<- unsigned long int Maths::GetFrequencyOfPrimes() const
<- int main(int,char **)
double Sqrt(double)
<- unsigned int FrequencyOfPrimes(unsigned int)
<- int main(int,char **)

(...)

Function: double Sqrt(double) (functions.cpp:3), calls: 1999997,
total ticks: 1470000, total user time: 900000ms, total kernel time: 396000
Function: int main(int,char **) (main.cpp:4), calls: 1, total ticks:
7170000, total user time: 956000ms, total kernel time: 222000
```

Code 4-5 : Sortie textuelle du profileur avec une application de test

A titre d'exemple, la sortie textuelle du profileur a été partiellement reproduite sur le Code 5. La sortie du profileur est coupée en deux éléments. Dans un premier temps, on peut constater que le profileur donne les différents chemins d'appels vers la fonction `Sqrt()` (qui calcule la racine carrée d'un nombre). On peut ainsi constater qu'elle a été appelée à chaque fois par la fonction `FrequencyOfPrimes()` (qui permet d'énumérer les nombres premiers entre 1 et un entier maximum), néanmoins, pas dans le même contexte d'appel. C'est pour cela qu'on la retrouve deux fois. De la même façon qu'elle était présente deux fois dans l'arbre contextuel d'appels. Dans la deuxième partie du Code 5, on peut constater que les fonctions sont énumérées. A leur suite, on retrouve de façon brute leurs données de performance : le nombre d'appels qui ont été réalisés à la fonction, ainsi que le nombre de tics d'horloge qui ont été consommés durant toute l'exécution de la fonction lors des appels, avec les détails sur le temps passé en mode utilisateur en mode système. Pour cette dernière partie, les fonctions ont été remises « à plat », c'est-à-dire que ces informations de performance concernent tous les appels à la fonction, peu importe le contexte d'appel. Donc, dans le cas du Code 5, la fonction `Sqrt()` n'apparaît qu'une fois, malgré ses deux chemins d'appel.

Sur la Figure 2, on peut observer la sortie de `acprof` sur notre application de test, une fois dans `Kcachegrind`. On peut constater dans la colonne de gauche, que l'on retrouve bien nos

différentes fonctions de l'application de test. On y retrouve également le coût en temps utilisateur (utime). A droite (en haut), on retrouve la représentation classique de `Kcachegrind` avec l'emboitement des fonctions, qui représente leurs appels et le coût consommé, tandis qu'en bas, on trouve la représentation des appels dans l'application de test et leur coût associé. On peut ainsi constater que l'arbre contextuel d'appel est nécessaire dans le cadre du profileur pour pouvoir fournir ce type de sorties. Par ailleurs, on constate également que `Kcachegrind` s'appuie sur les coûts inclusifs pour pouvoir évaluer la proportion des coûts et l'emboitement correct des fonctions. Ceci confirme les choix de développement de notre profileur.

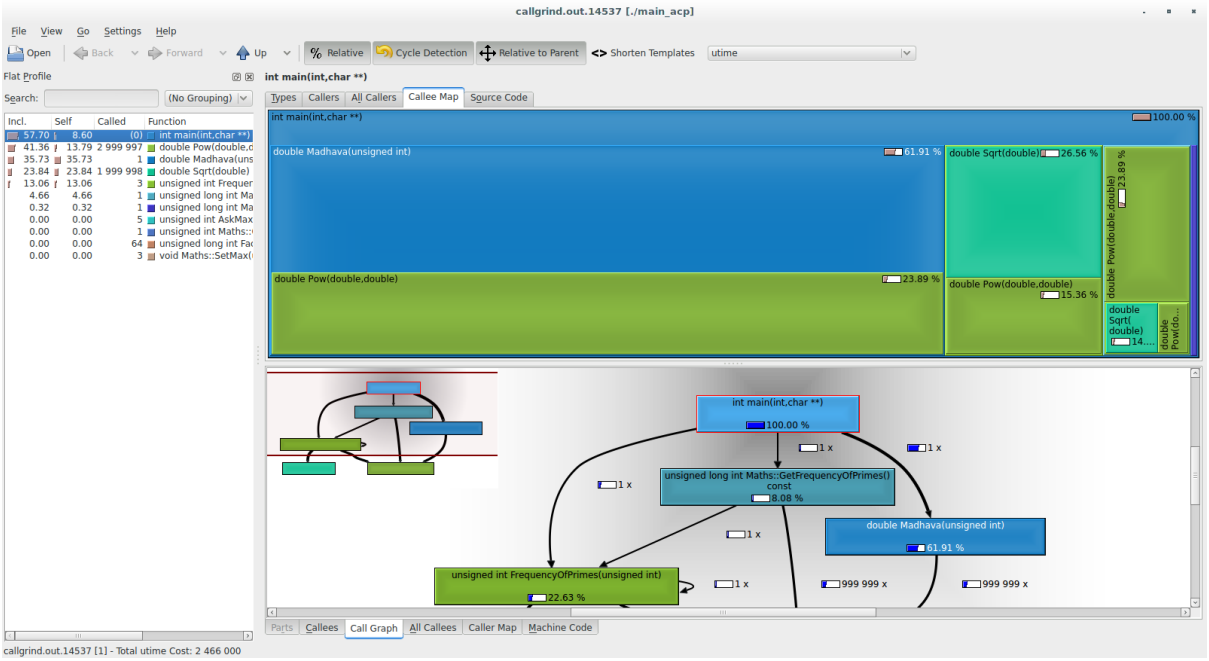

Figure 4-4 Affichage des données de performance une fois dans Kcachegrind

Le comportement du profileur étant conforme aux attentes, il a également fallu valider que les données de performance qu'il mesure correspondent effectivement à la réalité de l'application. Et qu'il n'y a pas de biais introduit, ou de mesure trop incorrecte. Pour ce faire, nous avons décidé de comparer le profilage d'une application témoin avec notre profileur et un autre profileur éprouvé. Nous avons donc choisi `callgrind`. Ce choix a été fait pour deux raisons. La première est que `callgrind` réalise un profilage asynchrone : il interrompt régulièrement l'application pour vérifier où elle en est dans son exécution et peut ainsi mettre à jour ses compteurs de performance. C'était une bonne opportunité pour confronter les deux approches de profilage. Par ailleurs, `callgrind` et notre profileur n'utilisent pas du tout les mêmes compteurs de performance. `callgrind` s'appuie sur l'utilisation des instructions processeur et de leur coût et peut alors conclure sur les performances, tandis que notre profileur ne s'intéresse qu'au temps utilisé par l'application.

Nous avons donc mis au point une seconde application de test très spécifique pour être capable de comparer les sorties des deux profileurs, alors qu'on ne considère pas les mêmes unités. Ainsi, nous avons mis au point une application qui comporte trois fonctions `f1`, `f2` et `f3`. Chacune de ces fonctions réalise la même opération mathématique en boucle, pendant un intervalle de temps connu et fixé, `n1`, `n2` et `n3`, respectivement. Cela signifie que la quantité de travail de chaque fonction est fixée à l'avance ainsi que son temps d'exécution. Cela nous permet non seulement d'estimer à l'avance quel sera le résultat du profilage mais également de pouvoir comparer la sortie des profileurs. Dès lors, nous avons fixé `n1` comme notre référence, puis `n2 = 2 * n1` et `n3 = 3 * n1`. Ainsi, nous pouvons émettre l'hypothèse que quelle que soit le profileur utilisé, la fonction `f3` représentera 50% du coût total, `f2` représentera environ 33,3% du coût total, et enfin, `f3` ne représentera qu'environ 16,7% du coût total. Une fois cette application de test implémentée, nous l'avons compilée deux fois : une fois brute sans l'aspect, ce sera la version qui sera donnée à profiler à `callgrind`, et une autre fois avec notre aspect de profilage. Pour le profilage avec notre profileur, nous avons utilisé le format de sortie compatible avec `Kcachegrind` afin de pouvoir avoir les résultats dans le même format, peu importe le profileur.

Tableau 4-2 Performances des fonctions dans l'application de test profilée avec `acprof` et `callgrind`

Fonction	Estimé	Aspect	Callgrind
f3	50%	49,99%	50,30%
f2	33,33%	33,33%	32,99%
f1	16,67%	16,68%	16,71%

Les résultats du profilage ont été reportés dans le tableau 2. On peut y constater que certes, les deux profileurs ne donnent pas exactement les mêmes résultats, néanmoins, on se retrouve avec des résultats extrêmement proches de ce qui était attendu. Ces résultats sont à mettre en perspective avec le simple fait que l'exécution d'un programme n'étant jamais identique, en raison de la charge du système, ces mesures changent légèrement d'une exécution à l'autre. Attendre des résultats strictement égaux serait donc un non-sens. On notera également que les variations sont de l'ordre de -/+ 0,50%, c'est-à-dire très peu.

Un autre résultat important a pu être mesuré lors de ce test : l'*overhead* du profileur. Ainsi, nous avons pu constater, en lançant plus de quatre fois le test, que notre profileur a un *overhead* inférieur à la seconde sur cette application de test. C'est à peine remarquable, et pourrait être confondu avec une variation « normale » du temps d'exécution. Dans tous les cas, cet *overhead* est largement

inférieur à celui de `callgrind`. Comme explicité plus haut, celui provoqué par `callgrind` est au minimum de 0,25X.

Ces deux applications de tests ont permis de valider le bon comportement et la cohérence de notre profileur. Nous nous sommes alors penchés sur la prise en charge par ce profileur des architectures parallèles qui sont au cœur de notre contexte de thèse.

3.4 – Prise en charge du parallélisme

Dans les applications s'exécutant sur des processeurs modernes, le parallélisme s'exprime au moins de deux façons totalement distinctes et ce avec des implications totalement différentes. On a tout d'abord le parallélisme de l'application elle-même qui lance plusieurs threads de calcul. D'autre part, on se trouve confronté également le parallélisme de l'architecture matérielle sur laquelle l'application s'exécute. Et quand bien même l'application serait purement séquentielle, elle sera affectée par ce dernier type de parallélisme. Depuis un peu plus d'une dizaine d'années, il s'agit de supporter les machines dites multi-cœurs, peu importe qu'ils soient logiques ou physiques. Le support du multi-cœur est relativement aisé à prendre en compte, il ne nécessite que de nouveaux compteurs, pas besoin de sections critiques ou de gestion atomique de variables.

Sur une machine multi-cœurs, une application, qu'elle soit séquentielle ou non, peut ne pas rester sur le même cœur durant toute son exécution. En effet, l'ordonnanceur du système, qui gère l'affectation des processus et des threads sur les cœurs ainsi que leur priorité d'exécution peut pour des raisons qui lui sont propres déplacer les processus d'un cœur à un autre. Typiquement, on retrouvera ces déplacements pour des raisons de gestion de charge : deux processus peuvent partager un cœur temporairement si l'un des deux processus est en attente d'entrée et que ça libère le cœur pour le second processus (Aas, 2005). Mais on peut également les retrouver pour des raisons plus simples de gestion de température. Si un processus tourne à 100% sur un cœur, et que la charge du système est basse, il peut être intéressant de faire passer le processus sur un autre cœur afin d'éviter un échauffement localisé du processeur et de conserver plus longtemps une fréquence d'exécution plus haute. Néanmoins, bouger une application d'un cœur à un autre a un coût, puisqu'il faut notamment déplacer les caches de plus bas niveau de l'application vers le nouveau cœur (aucune mesure précise ni rigoureuse n'a été trouvée jusque-là, cependant). Si un déplacement occasionnel n'a pas trop d'impact, une application qui passe son exécution à être déplacée de cœur en cœur verra ses performances réduites. Fort heureusement, si un profilage

fournit l'information qu'une application est trop souvent déplacée, il est alors possible de la lier à un ou plusieurs cœurs grâce au mécanisme d'affinité. Ce mécanisme permet de signaler au système que l'application ne doit s'exécuter que sur les cœurs qu'on autorise (et qu'on donne). Cela signifie notamment que si l'on ne donne qu'un cœur, elle n'en bougera pas. Pouvoir observer les déplacements d'une application sur les différents cœurs d'une machine grâce au profilage peut donc se révéler intéressant pour ses performances. C'est pourquoi nous avons décidé d'implémenter un tel support dans notre profileur par aspect. Par ailleurs, pour ce profilage, nous n'avons pas choisi l'hypothèse simplificatrice d'une machine homogène. Sous cette hypothèse, cela signifie que le changement de cœur n'a pour seul effet de provoquer une petite perte de performance pour la migration. Il n'y a en effet pas de changement de performance lié à la différence de performance entre les deux cœurs. Sans cette hypothèse, qui est donc notre choix de travail, il faut partir du principe que l'on peut travailler sur une machine hétérogène. C'est-à-dire que les cœurs ne sont pas forcément égaux. On peut se retrouver sur une machine avec l'hyper-threading et donc avec un cœur logique qui n'a pas la même performance qu'un cœur physique (Leng, Ali, Hsieh, Mashayekhi, & Rooholamini, 2002) ou bien sur une machine multi processeurs où les différents processeurs ne sont pas les mêmes, proposant des performances non égales. L'impact d'une machine non homogène sur les performances d'une application peut être réellement important (Balakrishnan, Rajwar, Upton, & Lai, 2005) en raison de l'hypothèse d'homogénéité faite par l'ordonnanceur du système d'exploitation (T. Li, Baumberger, Koufaty, & Hahn, 2007).

3.4.1 – Déplacement de l'application sur les cœurs

Pour pouvoir gérer le suivi des déplacements de l'application sur les différents cœurs du système, nous avons simplement étendu la structure `TCallInfo`, c'est-à-dire, l'arbre contextuel des appels, pour ajouter un compteur pour les passages sur les cœurs. Il a ici été stocké dans un « `vector` », comme montré sur le Code 6.

```
struct TCallInfo {
    ...
    std::vector<unsigned long> fCoreUsage;
};
```

Code 4-6 : Champ ajouté à l'arbre contextuel des appels pour gérer les déplacements sur les cœurs

Le tableau `fCoreUsage` aura au maximum une taille égale au nombre de cœurs sur la machine. La notion de cœur ici est totalement indépendante des notions de cœurs logiques ou de cœurs

physiques. En effet, le noyau Linux expose les cœurs de façon indifférenciée. Le profileur incrémentera la valeur dans la case correspondante au cœur actuel d'exécution à la fin de l'exécution de la fonction. Le numéro du cœur sur lequel l'application est en train d'être exécutée peut être récupéré grâce à l'appel système `sched_getcpu()`.

La contrepartie de cette récupération d'information est qu'elle ne peut pas être représentée au format compatible avec `callgrind`. De fait, nous avons rendu le profilage de cette information optionnel. Et lorsque l'utilisateur souhaite avoir cette information, alors le compte rendu de profilage est obligatoirement affiché sur la console, qu'importe l'option choisie par l'utilisateur. Pour pouvoir activer ce profilage, nous avons à nouveau étendu les options que supporte le profileur avec la ligne de commande avec l'option « `--acprof-with-core=usage` ». Sur le Code 7, nous avons reproduit partiellement la sortie du profileur avec cette option. Nous avons notamment ôté la première partie et la deuxième partie du profilage, qui contiennent comme précédemment, la liste des chemins d'appel des fonctions, ainsi que les coûts des appels des fonctions. En revanche, une troisième partie a été ajoutée, et c'est celle qui est reproduite sur le Code 7. Celle-ci contient la liste des fonctions, leur nombre d'appels et le numéro du cœur de calcul sur lequel elles ont fini de s'exécuter.

```
Function: double Sqrt(double)
Core 0: 999998 calls
Core 7: 999999 calls
Function: int main(int, char **)
Core 0: 1 calls
Function: unsigned int AskMax(const std::basic_string<char> &)
Core 0: 1 calls
Core 7: 4 calls
```

Code 4-7 : Extrait de la sortie du profileur avec l'option permettant de spécifier l'utilisation des cœurs

On peut constater avec l'extrait du Code 7, que l'application a bien été déplacée durant son exécution. On peut émettre l'hypothèse qu'elle s'est retrouvée sur deux cœurs différents : le 0 et le 7. Néanmoins, si l'application a été déplacée durant l'exécution d'une fonction, ce profilage sera incapable de le dire. Ce profilage s'il est simple et rapide, manque néanmoins de précision.

3.4.2 – Déplacement des fonctions sur les cœurs

Nous avons donc cherché à améliorer ce profilage, pour pouvoir également fournir une indication sur le cœur d'exécution au début de la fonction. Pour rappel, étant donné que notre profileur est purement synchrone, il ne peut pas être plus précis que cela. Il n'est pas en mesure de détecter les changements de cœurs qui surviendraient en plein milieu de l'exécution de la fonction, s'ils surviennent plusieurs fois. Pour ajouter ce support du stockage du cœur de début et de fin et d'exécution, nous avons à nouveau modifié notre arbre contextuel d'appel pour qu'il stocke plus d'informations sur les appels. Nous avons rajouté deux champs, comme montré sur le Code 8.

```
struct TCallInfo {
    ...
    unsigned long * fCoresTracks;
    unsigned long   fNbProcs;
};
```

Code 4-8 : Modifications apportées à la structure `TCallInfo` pour supporter le stockage du cœur en début et en fin d'exécution

Ces deux champs vont permettre de stocker le cœur de départ et le cœur de fin, tout en permettant de les corréler pour suivre précisément le déplacement de l'application au cours de l'exécution d'une fonction. Le second champ, `fNbProcs` permet uniquement de stocker le nombre de cœurs maximum qui a été observé sur la machine à un instant t. Cette valeur est nécessaire pour la gestion correcte de `fCoresTrack`. Ce champ permet le stockage d'une matrice carrée de la taille du nombre de cœurs (la valeur de `fNbProcs`). Même s'il n'est pas déclaré comme un tableau standard de dimension 2 (typiquement `unsigned long **`), il va être utilisé de cette façon-ci. Cela permet juste d'optimiser son allocation et son empreinte mémoire, puisqu'on alloue un unique tableau de taille $\texttt{fNbProcs}^2$. La matrice étant juste représentée par sa succession de lignes dans le tableau. La matrice s'utilise alors de la façon suivante : étant donné une ligne `i` et une colonne `j`, la valeur de la matrice à l'index (`i`, `j`) valant `n`, cela signifie que la fonction, dans ce contexte d'appel (rappelons que l'on est dans l'arbre contextuel d'appel) aura bougé du cœur `i` au cœur `j` au cours de `n` appels. Elle aura donc été appelée au moins n fois. Et la somme des différentes valeurs dans la matrice correspond au nombre d'appels de la fonction en général, c'est-à-dire à la valeur de `fCalls`, dans `TCallInfo`. De même, la valeur d'une case (`i`, `i`) correspond au nombre d'appels dont le cœur de départ (`i`) et le même que celui d'arrivée. Le champ `fNbProcs` permet donc de pouvoir utiliser correctement cette matrice, pour connaître la taille d'une ligne, ainsi que sa taille globale. Nous avons fait ce choix, pour la même raison qui nous a conduit à utiliser un `vector` pour `fCoreUsage` plutôt qu'un tableau de taille fixe, alloué tout au début. En effet, une machine peut

avoir un nombre de cœurs logiques exposés variable. Il est tout à fait possible d'éteindre et de démarrer des cœurs en pleine exécution de la machine, et donc des applications dessus. Dès lors, il devient nécessaire que dans le profileur, nous puissions nous adapter à cette géométrie variable. Nous utilisons ainsi un `vector`, plus souple, et pour notre matrice, nous gardons toujours sa taille pour pouvoir la faire évoluer en cas de besoin.

Avec une telle approche s'est alors posée la problématique de la gestion du stockage du cœur de départ, puisque l'approche matricielle impose de connaître et le cœur de départ et le cœur d'arrivée. La solution à cette problématique a été de stocker le cœur de départ, `i`, dans `fCoreUsage` à l'index 0. En effet, avec le suivi des cœurs à la fois au début et à la fin de l'appel, `fCoreUsage` n'est plus utilisé. On peut donc s'en servir comme une variable temporaire. Dès lors, une fois à la fin de l'exécution de la fonction, il suffit de récupérer la valeur à l'index 0 de `fCoreUsage` (`i`), le cœur actuel (`j`), et d'incrémenter la valeur dans la matrice `fCoreTracks` à l'index (`i` , `j`) de 1.

En effet, nous avons fait le choix de conserver la fonctionnalité de simplement afficher les cœurs en fin d'exécution, en parallèle de cette fonctionnalité plus précise, et donc de conserver le champ `fCoreUsage`, en raison de la potentielle consommation mémoire de cette fonctionnalité. Si nous considérons une application scientifique tournant sur un nœud de calcul avec 140 cœurs, on se retrouvera donc avec une structure `TCallInfo` qui aura au moins une taille en mémoire de 153,125 ko (140^2 x 8 octets – qui correspond à la taille d'un `unsigned long` sur un Linux 64 bits), ceci ne correspondant qu'à la matrice stockée dans la structure. De plus, si nous faisons également l'hypothèse que nous aurons 1000 fonctions appelées selon des chemins d'appels différents, c'est-à-dire, 1000 structures `TCallInfo` différentes dans l'arbre contextuel d'appel, on se retrouve avec déjà quasiment 150 Mo utilisés par le simple stockage de cette information. Si l'application est fortement consommatrice de mémoire, il peut être nécessaire de réduire l'empreinte mémoire du profileur, tout en permettant d'avoir néanmoins une idée des déplacements de l'application sur les cœurs. C'est cette constatation qui nous a conduits au choix de garder les deux options de profilage du parallélisme dans le profileur. Ainsi, pour pouvoir sélectionner cette fonctionnalité, nous avons étendu la ligne de commande. Il suffit d'utiliser l'option « `--acprof-with-core=track` ». L'utilisation de « `--acprof-with-core=usage` » permettant de ne suivre que le cœur de début de la fonction, comme explicité précédemment.

Ici, nous nous sommes heurtés au même problème qu'avec la mise en forme cœurs en fin d'exécution : il n'était pas possible de les exprimer de façon cohérente dans la syntaxe des fichiers de sortie `callgrind`. Nous avons donc également forcé l'utilisation de la sortie textuelle sur la console. Une sortie partielle est reproduite sur le Code 9.

```
Function: double Madhava(unsigned int)
Core 1 to core 5: 1 calls
Function: double Pow(double,double)
Core 1 to core 1: 2535 calls
Core 5 to core 5: 997464 calls
Function: double Sqrt(double)
Core 1 to core 1: 894046 calls
Core 4 to core 4: 105952 calls
Core 5 to core 5: 999999 calls
Function: int main(int,char **)
Core 4 to core 5: 1 calls
Function: unsigned int AskMax(const std::basic_string<char> &)
Core 1 to core 1: 1 calls
Core 4 to core 4: 1 calls
Core 5 to core 5: 3 calls
Function: unsigned int FrequencyOfPrimes(unsigned int)
Core 4 to core 1: 1 calls
Core 5 to core 5: 1 calls
Function: unsigned int Maths::GetMax() const
Core 5 to core 5: 1 calls
Function: unsigned long int Factorial(unsigned char)
Core 5 to core 5: 1 calls
Function: unsigned long int Maths::GetFrequencyOfPrimes() const
Core 5 to core 5: 1 calls
Function: unsigned long int Maths::GetSum() const
Core 5 to core 5: 1 calls
Function: void Maths::SetMax(unsigned int)
Core 4 to core 4: 1 calls
Core 5 to core 5: 2 calls
```

Code 4-9 : Sortie partielle du profilage lors de suivi du déplacement des fonctions sur les cœurs au début et à la fin de l'exécution

Sur la sortie partielle du Code 9, on peut constater que l'application a bougé du cœur 4 vers le cœur 5 et qu'elle a fait un court passage sur le cœur 1. En revanche, cette sortie doit être interprétée avec

précautions, et des conclusions erronées ne doivent pas être tirées. En effet, il ne faut pas conclure que l'application n'a été que sur les cœurs 1, 4 et 5. En effet, même si les autres cœurs n'apparaissent pas dans la sortie, il est possible que l'application y soit allée très brièvement, lors de l'exécution d'une fonction, auquel cas, le synchronisme du profilage et la finesse du profilage de l'ordre de la fonction empêchent de le voir. De même, une fonction qui a le même cœur d'exécution au début et à la fin peut cependant avoir utilisé un autre cœur au cours de son exécution. Néanmoins, on peut constater que les fonctions où il y a eu changement de cœur n'ont vécu ce changement qu'une seule fois. Le profileur n'a donc observé que trois changement de cœurs. On peut donc émettre l'hypothèse prudente qu'a priori, l'application semblait stable sur ses cœurs, et n'a pas fait de nombreux va-et-vient. D'autant que le changement de cœur a notamment été détecté sur la fonction main, la plus longue. Ce qui permet de conclure, avec certitude grâce aux améliorations apportées, que l'application a démarré sur le cœur 4 et fini sur le cœur 5. Puis, le changement de cœur du 4 vers le cœur 1 aurait eu lieu lors de l'exécution de la fonction `FrequencyOfPrimes()`, puis que le changement du cœur 1 vers le 5 aurait eu lieu lors de l'exécution de la fonction `Madhava()`. Cela pourrait constituer une hypothèse du déroulement des mouvements de l'application.

Cette première étape de prise en charge du parallélisme étant complète, nous sommes passés à l'étape suivante. En effet, cette première étape n'était là que pour prendre en compte le parallélisme lié aux architectures multi-cœurs des processeurs modernes. Elle ne prend pas en compte le parallélisme de l'application. Cette prise en charge est effectuée dans une seconde étape. Comme expliqué dans l'état de l'art, au sein d'une application, le parallélisme peut prendre plusieurs formes. L'application de simulation peut, par exemple, être distribuée. Auquel cas, profiler toutes les instances n'est d'aucune utilité. Etant donné que le même code est distribué, il suffit de profiler une des instances pour avoir les informations de performance sur l'application en général. De même, si l'application est parallélisée *via* des appels à la fonction `fork()` (multi-processus), alors les processus fils seront considérés comme une nouvelle application à part entière et seront profilés correctement. Dans le cas d'une application parallélisée grâce à OpenMP, d'autres outils de profilage existent déjà, nous ne les traitons pas pour notre profilage. Nous nous focalisons plus particulièrement sur les applications parallèles utilisatrices de la bibliothèque `phtread`, celles qui s'appuient donc sur les threads POSIX.

3.4.3 – Prise en charge du parallélisme avec les threads POSIX

La première problématique pour le support des applications parallèles est la protection de l'arbre contextuel des appels. En effet, plus rien ne garantit que l'arbre sera appelé de façon séquentielle, et qu'il sera donc dans un état cohérent au moment de l'appel. D'autant plus qu'à chaque utilisation de l'arbre, on est soit en début d'exécution, soit en fin d'exécution d'une fonction, c'est-à-dire à un moment de l'exécution où l'on va mettre à jour l'arbre : ajout d'une fonction fille, augmentation du nombre d'occurrence d'une fonction, mise à jour des compteurs, etc. L'arbre est donc à considérer comme une ressource critique. Il est donc nécessaire de s'assurer qu'un seul accès ne sera fait à l'arbre, d'où le besoin d'une exclusion mutuelle. Dès lors, les accès à l'arbre seront séquentiels, et le profileur sera « *thread-safe* » dans le cadre des applications parallèles. Cette utilisation d'un mécanisme de verrouillage de l'arbre est possible avec un *overhead* extrêmement faible du profileur : on passe peu de temps dans son propre code, les risques d'attente importante sont donc extrêmement faibles, et l'impact sur les performances sera donc limité. Nous avons donc ajouté dans notre code un mutex, comme montré sur le Code 10.

```
pthread_mutex_t fBigLock;
```

Code 4-10 : Définition du mutex utilisé dans le profileur

Comme visible sur le Code 10, ce mutex est appelé « *big lock* ». Ce choix de dénomination a été fait par analogie à un mécanisme de verrouillage dans le noyau Linux, le « *Big Kernel Lock* ». Ce verrouillage, très particulier, permettait de verrouiller l'intégralité du noyau pour les exécutions concurrente sur une machine multi-cœur. Ceci bloquait l'intégralité du noyau pour une unique tâche. Il a depuis été supprimé, le noyau Linux étant maintenant réentrant. Dans le cadre du profileur, on retrouve le même principe de « *big lock* ». Dès qu'une fonction entre dans la partie du profileur, elle verrouille l'intégralité du profileur grâce au mutex, afin d'assurer d'être la seule à le manipuler et à le lire. Cette approche sans finesse pour le verrouillage n'est probablement pas la plus souple, néanmoins, elle fournit une solution simple, efficace et sûre pour s'assurer qu'il n'y aura pas de compétition dans le profileur. Etant donné la structure du profileur, une approche plus fine requerrait plus d'ingénierie pour un faible rendement étant donné le faible *overhead* du profileur.

On peut également constater sur le Code 10, que notre mutex est un objet `pthread`. Si cela semble cohérent, étant donné que l'on souhaite profiler des applications parallèles s'appuyant sur `pthread`, cela pose néanmoins un problème dans le cadre des applications séquentielles qui n'auraient pas cette dépendance à `pthread`. En effet, il est inutile d'imposer cette dépendance à l'utilisateur pour une application séquentielle, alors que l'arbre ne sera même plus une section

critique, rendant l'usage du mutex inutile. Nous avons donc ajouté une directive pour le préprocesseur comme montré dans le Code 11.

```
#ifdef ACPROF_WITH_MULTITHREAD
```

Code 4-11 : Directive préprocesseur pour délimiter la dépendance à `pthread`

Le Code 11 est la directive à utiliser avec le préprocesseur du compilateur pour délimiter les zones de codes qui sont nécessaires pour la prise en charge des applications *multi-threadées*, et notamment la gestion de la section critique. Dans le cas où un utilisateur compile une application *multi-threadée*, il doit alors fournir le paramètre de compilation « -`DACPROF_WITH_MULTITHREAD` » afin que le code soit activé dans le profileur. Autrement, le profileur ne verra qu'une application parallèle, et sera sujet à la corruption de ses données internes. Cette méthode assure donc qu'aucune dépendance non désirée n'est ajoutée au programme de l'utilisateur final.

Néanmoins, ce simple changement ne règle pas tous les problèmes causés par le parallélisme de l'application, et n'empêche toujours pas totalement la corruption de l'arbre contextuel des appels. En effet, si on revient au support des architectures multi-cœurs, il était question de stocker le cœur de départ de la fonction dans `fCoreUsage[0]`. Dans le cas séquentiel, cette stratégie ne pose, évidemment aucun problème. Dans le cas parallèle, en revanche, on se retrouve avec une corruption de l'arbre. En effet, dans le cas où deux fonctions parallèles débuteraient en même temps, elles écriraient toutes les deux leur cœur de départ au même endroit. Une des données serait perdue. Pire encore, si une troisième fonction est démarrée pendant que les deux autres tournent, la valeur sera à nouveau écrasée. Et il en résulte une matrice des mouvements de fonctions sur les cœurs qui n'aura plus aucun sens, et qui de surcroît sera fausse. Afin de régler ce problème, la solution d'exploiter l'aspect vectoriel de `fCoreUsage` a été choisie. En effet, étant donné que `fCoreUsage` est un tableau, il est possible d'y stocker plusieurs valeurs. On peut donc stocker plusieurs cœurs de départ, pour plusieurs fonctions, en les indexant sur leur identifiant de thread. Car, pour un thread donné, on retrouve bien une exécution séquentielle, il n'y a donc plus aucun risque de corruption des cœurs de départ. Néanmoins, ici, afin d'avoir un fonctionnement correct, l'identifiant de thread considéré ne sera pas l'identifiant système, mais un identifiant propre au profileur, indexé à partir de 0 et étant contigu. On notera donc également que ce modèle est totalement compatible avec une application séquentielle. En effet, dans une application séquentielle, on ne retrouve qu'un seul processus, qu'un seul thread, qui sera donc le thread d'identifiant 0, et son cœur de départ sera donc écrit dans `fCoreUsage[0]`, comme avant.

La question de l'identification des threads se pose donc, afin de pouvoir leur assigner un identifiant unique. Pour ce faire, nous avons ajouté un `set`, issue de la Standard Template Library (STL), pour stocker les threads, comme montré sur le Code 12.

```
std::set<pthread_t, TThreadsCompare> fThreadsId;
```

Code 4-12 : `set` de la STL pour stocker les identifiants de threads

On peut constater deux éléments importants sur le Code 12. Dans un premier temps, on s'appuie toujours sur l'identifiant de la bibliothèque `pthread` pour pouvoir identifier nos threads, la correspondance avec notre identifiant n'étant faite qu'en local. De plus, le set est ordonné selon un ordre précis, grâce au foncteur `TThreadsCompare`, montré dans le Code 13.

```
struct TThreadsCompare {
    bool operator() (const pthread_t & t1, const pthread_t & t2) const {
        return (pthread_equal(t1, t2) != 0);
    }
};
```

Code 4-13 : Foncteur pour classer les threads

Ce foncteur, qui prend donc en paramètres deux identifiants de threads issus de la bibliothèque `pthread`, s'assure qu'ils ne référent pas au même thread, grâce à un appel à la fonction servant cette fonctionnalité : `pthread_equal()`. Celle-ci, directement issue de la bibliothèque `pthread` permet de comparer deux objets `pthread_t`. En effet, il n'est pas possible de comparer deux objets `pthread_t` directement. Leur type est opaque et il n'y aucune garantie qu'il reste le même dans le temps. Il est donc interdit d'avoir directement recours aux opérateurs de comparaison par défaut du langage C++. C'est notamment la raison pour laquelle il est nécessaire d'implémenter un tel foncteur et de le fournir au `set`. Autrement, le `set` ferait une comparaison brute qui ne présenterait aucun intérêt et ne produirait pas le bon résultat. Ces deux bases étant données, il suffit dès lors d'écrire une fonction qui permet de récupérer l'identifiant du thread pour notre profileur. Il apparaît évident que le set `fThreadsId` est à nouveau une ressource critique. Néanmoins, la STL ne fournit pas d'implémentation de ses structures de données qui soient « *thread-safe* » (Plauger, Lee, Musser, & Stepanov, 2000). Il est donc nécessaire qu'avant d'accéder au set, le « *big lock* » du profileur soit verrouillé, pour empêcher toute situation de compétition

entre threads. La fonction décrite dans le Code 14 doit donc impérativement être appellée avec le mutex verrouillé.

```
unsigned long GetThreadId() {
    pthread_t tId;
    std::set<pthread_t, TThreadsCompare>::iterator currentThread;
    tId = pthread_self();
    currentThread = fThreadsId.insert(fThreadsId.end(), tId);
    return std::distance(fThreadsId.begin(), currentThread);
}
```

Code 4-14 : Implémentation de la fonction qui permet de retourner un identifiant unique de thread

La fonction `GetThreadId()` présentée sur le Code 14 est très simple, et s'appuie sur les propriétés de la bibliothèque `pthread` et du container `set` de la STL pour assurer l'unicité de l'identifiant. En effet, l'identifiant retourné par `pthread_self()` est unique d'un point de vue `pthread`. Deux threads ne peuvent pas avoir le même (sauf si ils sont effectivement le même thread). De son côté, le container `set` assure qu'un seul élément peut être mis dedans. Si deux sont identiques, alors un seul sera gardé. C'est pourquoi, le foncteur du Code 13 est important, c'est lui qui permet au `set` de correctement assurer l'unicité des éléments qu'il contient. Ainsi, la fonction `GetThreadId()` se contente de récupérer l'identifiant du thread de `pthread` grâce à `pthread_self()` puis tente de l'insérer dans le set grâce à la fonction `::insert()` du `set`. Cette fonction tente d'insérer la valeur dans le `set`. Si elle échoue à l'insérer parce qu'il est déjà présent, elle retournera sa position actuelle dans le `set`. Si au contraire, il n'est pas présent, elle l'insèrera et retournera la position d'insertion. Dans tous les cas, donc, la fonction retourne la position de l'élément dans le `set`, peu importe qu'il ait été ajouté ou non. La position de l'élément dans le `set` est un itérateur. C'est un type de la STL qui pointera toujours vers le même élément, tant que les éléments du container ne sont pas changés. Dans le cas présent, l'intérêt est qu'il est extrêmement facile d'avoir un itérateur pointant vers le début du `set`, grâce à la fonction `::begin()`. Dès lors, l'utilisation de `std::distance()` permet de faire la différence entre les deux itérateurs, c'est-à-dire de connaitre la distance qui les sépare. Ceci nous donne donc un identifiant unique qui nous est propre pour chaque thread. Vu que si deux points sont à la même distance d'un autre sur une même droite, c'est qu'ils sont confondus. Par ailleurs, on constate également qu'il est facile de borner cet intervalle. Il sera compris entre 0 (1 thread, sa distance à `begin()` est nulle, il est `begin()`) et le nombre maximum de threads lancés.

Néanmoins, l'un des problèmes de `AspectC++` est son incapacité à pouvoir insérer des *advices* autour de fonctions issues de bibliothèques. Donc, typiquement, autour de `pthread_create()` dans notre cas. Nous avons en effet besoin de pouvoir tisser (« *weave* » en anglais orienté aspects) autour de `pthread_create()`. Cela signifie que pour que l'utilisateur ait des résultats cohérents, il est malheureusement nécessaire qu'il modifie légèrement son code pour changer ses appels à `pthread_create()` vers une fonction statique « *wrapper* » définie dans le Code 15.

```
#if defined _PTHREAD_H
int TProfiler_pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                             void *(*start_routine) (void *), void *arg) {
   return pthread_create(thread, attr, start_routine, arg);
}
#endif
```

Code 4-15 : Fonction `pthread_create()` du profileur pour lancer un nouveau thread

Comme on peut le constater sur le code 15, cette fonction ne fait que transmettre l'appel à `pthread_create()` dont elle partage la signature. La seule utilité de cette fonction est donc de permettre d'écrire des *advices* pour elle. Néanmoins, on peut également constater que cette fonction, pour pouvoir être appelée dans le code, sans briser la compilation a dû être sortie de « l'aspect ». Dès lors, cela signifie que l'aspect vient avec un fichier .cpp supplémentaire qui contient l'implémentation de cette fonction. Nous nous sommes néanmoins arrangés pour que ce fichier puisse tout le temps être utilisé lors de la compilation : si le fichier d'en-tête des threads n'est pas inclus (ie, compilation sans le support des threads), elle n'est tout simplement pas définie, d'où le `#ifdef`. Selon l'usage du fichier .cpp fait par l'utilisateur, il pourra être nécessaire qu'il définisse le prototype de la fonction dans l'un de ses fichiers d'en-tête.

Ceux-ci sont en effet nécessaires afin de pouvoir préserver la cohérence de l'arbre contextuel des appels. En effet, il est nécessaire que nous sachions quand une fonction démarre un nouveau thread et à quel point dans l'arbre contextuel des appels elle est. Nous devons donc stocker ces deux informations avant le début du thread. Plus exactement, on stockera : la fonction qui démarre le thread (celle qui appelle `TProfiler_pthread_create()`), ainsi que le niveau de la fonction dans l'arbre. Le niveau correspond simplement au nombre d'appels qui permettent d'atteindre la fonction depuis la fonction `main()`. On peut voir cela comme le nombre de fonctions parentes, grand parentes,… que possède une fonction. Le thread créé peut donc récupérer ces informations. Néanmoins, ceci a nécessité quelques changements dans le profileur. Ces deux champs étaient

toujours présents dans le profileur, sous les noms respectifs `fParent` et `fLevel`. Ces deux champs sont toujours utilisés de la même façon, pour que le profileur puisse savoir où chercher/insérer le nouveau `TCallInfo` en cas d'appel à une nouvelle fonction ; ils représentent l'état du profileur. Il s'agissait donc de rendre cela possible pour plusieurs threads. On voit bien que deux simples variables ne suffisent plus pour garder l'état de plusieurs threads. La solution ici est à nouveau de transformer ces deux variables en vecteurs. Ainsi, chaque case du vecteur correspondra à un thread. L'identifiant partant de 0, cela rend l'usage plus facile du vecteur. On utilise un vecteur et non un tableau fixe ici, la taille du vecteur pouvant varier énormément : on ne sait pas a priori, combien de threads seront lancés. En plus de ces changements, il était nécessaire de rajouter des champs supplémentaires. En effet, une fois qu'un thread démarre, il ne connait pas ces deux informations, il n'est relié à rien. Ces informations n'étant stockées qu'une fois le thread démarré. Nous avons donc créé un *advice* pour la fonction `pthread_create()` du profileur qui récupère ces informations avant de lancer le thread (avant d'appeler la fonction `pthread_create()` de la bibliothèque `pthread`, donc). Ces données doivent être stockées de façon statique hors de tout contexte, pour que le thread, une fois lancé, puisse les récupérer. Bien évidemment, il est également nécessaire de les protéger, afin qu'un nouveau thread ne puisse pas les effacer avant que le thread venant d'être lancé ne les ait récupérées. Cela implique la création d'un nouveau mécanisme d'exclusion mutuelle. Les quatre champs définis dans le Code 16 ont donc été rajoutés.

```
TCallInfo*      fThreadCreator;
unsigned int    fCreationLevel;
pthread_mutex_t fCreateLock;
sem_t*          fReleaseLock;
```

Code 4-16 : Champs ajoutés au profileur pour la création de threads

Le champ `fThreadCreator` permet donc de stocker la fonction qui a initié la création du thread dans son contexte d'appel. C'est donc juste un pointer vers un nœud de l'arbre contextuel d'appel, afin de pouvoir le continuer. Le champ `fCreationLevel` permet de stocker le niveau de la fonction qui a initié la création du thread. Le champ `fCreateLock` est le verrou d'exclusion mutuelle, qui permet de s'assurer que les deux champs du dessus ne seront accédés que par un thread à la fois. Il est donc maintenu par la fonction parente. Le dernier champ, le sémaphore `fReleaseLock` permet au thread fils et à la fonction mère de communiquer et d'indiquer quand la fonction parente doit libérer le verrou `fCreateLock`, c'est-à-dire, une fois que les champs `fThreadCreator` et `fCreationLevel` ont été lus. La mise en place de tout ceci se déroule comme montré sur le Code 17.

```
advice execution(thread_create()) : around() {
    pthread_mutex_lock(&fCreateLock);
    fThreadCreator = GetParent();
    fCreationLevel = GetLevel();
    sem_init(fReleaseLock, 0, 0);
    JoinPoint::proceed();
    sem_wait(fReleaseLock);
    sem_destroy(fReleaseLock);
    pthread_mutex_unlock(&fCreateLock);
}
```

Code 4-17 : *Advice* pour la fonction `pthread_create()` du profileur

Comme visible sur le Code 17, la première chose à faire est donc de verrouiller `fCreateLock` pour être sûr de n'être que le seul thread à démarrer. Autrement, la fonction appelante sera mise en attente jusqu'à disponibilité des champs pour pouvoir démarrer un thread. Ainsi, une fois `fCreateLock` verrouillé, la fonction parente peut récupérer les informations pour le lancement grâce aux fonctions `GetParent()` et `GetLevel()` qui retournent les informations idoines. Enfin, le sémaphore de communication est initialisé pour que le thread créé puisse l'utiliser. Une fois cette partie réalisée, la création du thread peut débuter. C'est `JoinPoint::proceed()` qui s'en charge. Cet appel, propre à `AspectC++` permet de dire au compilateur `AspectC++` que l'appel à la fonction autour de laquelle il tisse doit débuter à cet endroit. Donc, techniquement, dans notre cas, c'est ici qu'aura lieu l'appel à `pthread_create()` de la bibliothèque `pthread`. Une fois qu'on a fini l'appel à `JoinPoint::proceed()` cela signifie que le thread a été démarré. On se place donc en attente sur le sémaphore. Celui-ci sera libéré par le thread créé. Dès lors, on pourra le supprimer et libérer le verrou d'exclusion mutuelle `fCreateLock`. Dans cet ordre-ci, on s'assure que les données ont bien été lues par le thread fils, puisqu'il n'autorise la levée du verrou qu'une fois qu'il les a lues.

Quand on regarde du côté du fils, dès qu'il est lancé, comme cela correspond à l'appel d'une fonction, celle-ci est profilée, donc, on retombe dans l'aspect de profilage. La première chose que fait l'*advice* est de récupérer l'identifiant du thread grâce à la fonction `GetThreadId()` ; le verrou `fBigLock` est donc déjà verrouillé également. Si cet identifiant n'a jamais été rencontré, cela signifie que l'on est dans un nouveau thread qui vient d'être démarré. Dès lors, dans l'*advice*, il est possible de récupérer les données des champs `fCreateLevel` et `fThreadCreator`. En effet, le verrou `fCreateLock` est toujours tenu par la fonction (et le thread) parent. L'accès aux données est donc cohérent. Une fois ces données remises dans l'arbre, la fonction ayant pu se situer dans

l'arbre contextuel d'appel, l'*advice* libère le sémaphore `fReleaseLock`, qui va donc conduire à la libération du verrou `fCreateLock` chez la fonction père, et donc, à la possibilité de lancer un nouveau thread. Pendant ce temps, le déroulement de l'*advice* continuera comme pour toute fonction, pour initialiser la prise de données de performances.

3.5 – Considérations post-implémentation

A l'issue du travail décrit dans les sections précédentes, le profileur est totalement capable de profiler une application parallèle sur un processeur multi-cœur. Néanmoins, les informations que fournit le profileur pour une application parallèle sont extrêmement limitées et ne sont en réalité que l'extension de ce qui se faisait déjà pour une application séquentielle : performances (temps d'exécution, etc.) et contexte d'appel. On ne retrouve aucune information spécifique au parallélisme lui-même. Dès lors se pose la question de savoir s'il faut fournir plus d'informations pour les applications parallèles, spécifiques à la gestion des threads. Le cas échéant, quelles informations supplémentaires surveiller et fournir ? Cela pose la question plus générale de la représentation des données. Nous avons pu constater que le format de sortie `callgrind` et l'application `Kcachegrind` elle-même ne sont pas réellement prévus pour afficher ou mettre en forme des données issues du profilage d'une application parallèle. L'ajout d'informations supplémentaires pourrait conduire à une mauvaise interprétation des données. D'où notre usage de la console, mais cela réduit la quantité d'informations que l'on peut fournir à l'utilisateur, et les capacités de mise en forme (simples) sont limitées également.

Dans le cas du parallélisme de l'architecture, une autre information pourrait être considérée et prise en compte dans le profilage : le nombre de « *context switches* » durant l'exécution d'une fonction. Un « *context switch* » (changement de contexte) se produit quand le système d'exploitation va, en cours d'exécution, changer le cœur de calcul affecté à l'application. En effet, un cœur est purement séquentiel et ne peut faire tourner qu'une application à la fois. Pour produire un fonctionnement multitâche, un système d'exploitation doit donc régulièrement interrompre l'application qui s'exécute sur un cœur pour la remplacer par une autre qui était en sommeil en attente d'un cœur disponible. Cette étape où une application est remplacée par une autre sur un cœur de calcul précis est appelée un « *context switch* ». Durant cette étape, le système d'exploitation va notamment remettre à zéro le pipeline du cœur, ainsi que ses caches spécifiques et TLB (Stravers & Van De, 2004). Le changement de contexte a donc un coût en performance. Celui-ci a été caractérisé dans (C. Li, Ding, & Shen, 2007) et est globalement dépendent de la taille des données en cache et de leur capacité à tenir en cache. Plus une application est consommatrice de données, plus son changement

de contexte est coûteux. Le nombre changements de contexte qui se produisent pendant l'exécution de l'application, et même pendant l'exécution de ses fonctions peut alors devenir une information utile pour les performances : plus il y a de changements de contexte, moins l'application sera performante et rapide. Les changements de contexte nombreux peuvent notamment avoir deux raisons principales. La première raison peut être que l'application est souvent en attente pour des entrées/sorties. Dans ce cas, le système d'exploitation va la mettre en attente et mettre à disposition le cœur pour une application qui a réellement des données à traiter. La seconde raison peut être que la machine est surchargée et doit faire tourner trop d'applications en même temps, auquel cas, de nombreux changements de contexte auront lieu pour que toutes les applications puissent accéder à un cœur. Malheureusement, il n'y aucune méthode « évidente » pour récupérer le nombre de changements de contexte, aucun appel système ne retourne cette information directement. Il est néanmoins possible de récupérer cette information *via* le pseudo système de fichiers « `procfs` » grâce au fichier « `status` » qui est disponible pour chaque processus. Celui-ci contient de nombreuses informations sur le processus actuellement lancé sur une machine, y compris le nombre de changements de contexte volontaires (attente d'entrée/sortie, par exemple) et involontaires (une autre application doit tourner, par exemple). *Via* une utilisation de ce fichier, il serait possible d'estimer le nombre de changements de contexte qui ont eu lieu lors de l'exécution d'une fonction, et donc, du processus profilé.

3.5.1 – Limites du profileur

Ceci nous amène aux limites du profileur *via* l'aspect (et sans bibliothèque externe). Comme expliqué précédemment, le profilage par aspect nous limite au profilage synchrone. Un profilage asynchrone n'aurait aucun intérêt à être fait avec de l'aspect. L'autre limite est que le profilage par aspect requière à chaque fois une recompilation du logiciel. On ne peut pas juste lancer l'application dans le profileur pour avoir le résultat comme avec `valgrind`, par exemple. Cela nous rapproche de comportements tels que celui de `gprof` où une recompilation est nécessaire. A noter que la recompilation est nécessaire avec `gprof`, justement parce qu'il rajoute du code pour pouvoir récupérer des performances dans le code de l'utilisateur qu'il compile.

Par ailleurs, la technologie utilisée pour pouvoir implémenter le profileur par aspect, `AspectC++`, est encore jeune et pas complètement aboutie, et donc, cela signifie que le profileur se retrouve avec les limites de l'outil lui-même. Par exemple, `AspectC++` ne supporte pas le code écrit en C++11. De même `AspectC++` ne supporte pas les points de coupure sur des fonctions génériques

avec des *templates*, qui ne soient pas des fonctions membres d'une classe. C'est-à-dire que le point de coupure défini dans le fragment de Code 1, ne pourra jamais correspondre à une fonction avec *template*, quand bien même ce point de coupure est supposé correspondre à toutes les fonctions avec ou sans classe, avec ou sans *template*. C'est un bug connu dans `AspectC++`, mais cela peut conduire `AspectC++` à ne pas profiler certaines fonctions. De la même façon, `AspectC++` peut ne pas profiler certaines fonctions parce qu'elles sont à arguments variables. Ici encore, il les ignore purement et simplement pour les points de coupure.

L'autre limite importante, déjà soulevée dans les sections précédentes est l'impossibilité pour `AspectC++` d'appliquer des *advices* sur des fonctions issues de bibliothèques, de fonctions dont on a juste l'entête. Dans notre cas, cela signifie que le profilage ne peut être que partiel, il ne sera pas possible de profiler le temps passé dans les appels systèmes ou dans les appels à des fonctions de bibliothèques. Ceci réduit donc fortement la finesse de profilage, contrairement à ce que d'autres profileurs peuvent faire. La seule méthode pour pouvoir profiler les appels externes est d'encapsuler les appels aux fonctions *via* des « *wrappers* » qui se contentent d'appeler la fonction de base. C'est notamment ce que nous avons fait dans la section 2.4 pour pouvoir appliquer des *advices* sur la fonction `pthread_create()` ou encore dans la section 2.3, dans le Code 5, on peut voir apparaître une fonction `Sqrt()`, celle-ci n'est qu'un *wrapper* autour de la fonction `sqrt()` de la bibliothèque standard. Mais cette fonction présentant un intérêt d'un point de vue performances, il devient alors intéressant de la profiler, d'où l'encapsulation.

Une autre limite inhérente à `AspectC++` est le niveau auquel on peut appliquer les aspects. Les points de coupure ne peuvent correspondre qu'à des fonctions, ainsi, notre profileur ne peut pas avoir une finesse de l'ordre de la ligne de code, voire de l'instruction en assembleur. Ce qui est forcément limitant par rapport aux autres profileurs.

Nos tests ayant démontré le fonctionnement attendu pour le profileur, il pourrait être nécessaire de reprendre les compteurs de performance et de proposer des compteurs plus précis afin de pouvoir établir des comparaisons plus fines avec les autres profileurs ; les compteurs actuels limitant cruellement la précision générale du profileur. On pourrait s'appuyer sur des bibliothèques externes, telle que Pin, d'Intel présentée plus haut, ou encore PAPI (Mucci et al., 1999).

Enfin, une dernière limite du profileur vient tout simplement de son implémentation en tant que telle, pour le support des applications *multi-threadées*. En effet, notre implémentation revêt deux

problèmes. Le premier que nous évoquions précédemment est qu'elle brise le principe de simple recompilation pour avoir l'aspect appliqué à l'intégralité du programme : il faut malheureusement changer des portions de code pour pouvoir prendre en compte le profilage des sections lancées dans des threads, ainsi que rajouter un fichier source. Ce qui amène au deuxième problème, seules les applications *multi-threadées* avec la bibliothèque `pthread` sont supportées. Tout autre forme de parallélisme sera « ignorée » par le profileur et conduira à une corruption des données dans l'arbre contextuel des appels. Néanmoins, cette limite peut être mitigée par le fait que pour profiler les autres formes de parallélisme, d'autres profileurs existent. Dans le chapitre d'état de l'art, nous avons proposé de nombreuses alternatives pour les applications s'appuyant sur d'autres formes d'API ou de bibliothèques pour fournir le parallélisme.

Finalement, à l'origine, nous avions développé ce profileur afin d'être capable de profiler notre simulation de Monte Carlo `tomusim` qui ne pouvait plus être profilée par `valgrind` suite à l'utilisation de la JNI. Malheureusement, deux facteurs ont conduit à ne jamais réaliser ce profilage. Dans un premier temps, `tomusim` a été abandonné au profit d'une nouvelle application, `tmvg4sim`, développée uniquement en C++, à l'aide de `Geant4` (Agostinelli et al., 2003). Cette nouvelle application étant parfaitement profilable avec `valgrind` voire avec `VTune`, il n'y avait plus aucune nécessité de la profiler avec notre profileur par aspect. Le second facteur est que les applications de test sur lesquelles fonctionnaient très bien le profileur ont un code source simple et court, pas assez proche des applications scientifiques réelles. Dans l'état actuel de notre profileur `AspectC++`, il n'a pas été possible de réussir à analyser le code de l'application `tomusim`. En effet, nous sommes plusieurs fois tombés sur des problèmes complexes de gestion du code à profiler. Il n'était même pas question de pouvoir compiler l'application à cette étape-ci.

Néanmoins, malgré ces deux derniers écueils qui nous ont empêchés d'arriver jusqu'au bout de notre processus, notre travail a permis d'établir qu'il est totalement possible de faire un profilage simple et léger par aspect en C++. Pour appuyer nos propos, nous avons été capables de fournir un prototype, développé à l'aide d'`AspectC++` qui permet de profiler une application séquentielle ou parallèle sur une architecture multi-cœurs. C'est ce prototype que nous avons donc testé sur la simulation de Monte Carlo plus légère et *multi-threadée*, `parsitomu`, que nous avons proposée dans la section 5 du chapitre 3 (Propositions) dont l'implémentation sera précisée dans le chapitre suivant.

3.5.2 – Profilage de `parsitomu` avec `acprof`

`parsitomu` est une simulation muonique de Monte Carlo légère implémentée en C++, et qui s'appuie sur une bibliothèque, PuMAS pour la propagation des particules. Celle-ci est décrite plus en détails au chapitre suivant. Contrairement à `tomusim`, elle ne fait appel à aucune autre dépendance et est parallèle, grâce à l'usage de la bibiothèque `pthread`. Elle rentre donc totalement dans les cibles de profilage pour `acprof`. Ainsi, faute de pouvoir tester le profileur sur l'application `tomusim`, nous avons décidé de profiler `parsitomu`.

Afin de pouvoir compiler notre simulation avec `acprof`, il nous a simplement fallu rajouter le prototype de la fonction `TProfiler_pthread_create()` dans l'un des en-têtes (celui de la « *threads factory* ») et de faire le changement de code visible sur le Code 18 ; à savoir, une compilation conditionnelle selon si oui ou non, le profileur avec support du multithreading est utilisé.

```
#ifdef ACPROF_WITH_MULTITHREAD
    int err = TProfiler_pthread_create(&fThreads[i], 0, ThreadHelper,
context);
#else
    int err = pthread_create(&fThreads[i], 0, ThreadHelper, context);
#endif
```

Code 4-18 : Utilisation de `TProfiler_pthread_create()` dans `parsitomu`

Avec le changement de code décrit dans le fragment 18, tout était prêt pour prendre en charge le parallélisme de l'application. Nous avons fait deux tests. Nous avons lancé la simulation avec le parallélisme standard dans un premier temps, et avec les « *pilot threads* » dans un second temps. Nous avons répété cette expérience deux fois : une fois avec la simulation compilée toute seule, la seconde fois avec le profileur compilé dans la simulation. Nous avons limité notre simulation à 100 évènements.

La première observation que nous avons pu faire est que le profileur marche correctement avec le parallélisme. Il n'y a pas de corruption de données, et les résultats sont cohérents avec notre connaissance de la simulation et ce que nous attendions comme résultats. On notera que les outils de surveillance du parallélisme (cf. section 4) ne repèrent aucune erreur lors de l'exécution. Néanmoins, autant l'exécution se passe sans encombre avec les « *pilot threads* », autant l'exécution

est complètement chaotique quand la simulation lance plus de cent threads. Le profileur peut difficilement maintenir l'exactitude de ses structures internes, en raison d'hypothèses faites sur les identifiants des threads qui ne s'avèrent pas toujours vérifiées. Cela implique que la simulation avec le parallèlisme non « *pilot threads* » se retrouve extrêmement instable avec le profileur et dans la majorité des cas (8 tentatives d'exécution sur 10), elle est finalement annulée en plein milieu d'exécution par le profileur. Toujours est-il que cela permet d'étudier le comportement du profileur en cas d'exécution parallèle.

On constate ainsi que sur une application réelle, contrairement aux applications de test, le profileur a un véritable *overhead*, non anticipé. Sur cinq exécutions (sauf dans le cas du parallélisme avec de nombreux threads), nous avons pu constater les données formulées dans le Tableau 3.

Tableau 4-3 : Temps d'exécution avec et sans profilage quel que soit le modèle de parallélisme

	Pilot threads	Parallélisme « standard »
Temps d'exécution seul	0,26 s	0,25 s
Temps d'exécution profilé	23 s	35 s

On constate sur le Tableau 3 qu'on a un *overhead* très important. Nous avons donc voulu le comparer avec celui de `valgrind`. Nous avons pu constater que `valgrind` a un *overhead* de l'ordre de 25X (le nôtre étant entre 88X et 140X). `valgrind` fournit donc de bien meilleures performances que notre profileur. Cependant, nous avons constaté un comportement surprenant avec `valgrind`. Notre simulation était lancée avec le paramètre lui autorisant quatre threads en parallèle. Ainsi, avec l'utilisation de la commande « `time` », on s'attend à une utilisation de CPU entre 300% et 400%, les threads étant répartis sur plusieurs cœurs (selon leurs possibilités d'exécution). Et c'est ce que l'on constate, d'ailleurs, lors d'une exécution normale (entre 320% et 380%). Avec notre profileur, nous constatons que ce taux reste stable, entre 350% et 360% (cette hausse pouvant être expliquée par la consommation supplémentaire induite par le profiler). Avec `valgrind`, toutes nos reproductions ont toujours abouti à 100%, exactement. Ce qui signifie que toute la simulation a été écrasée sur un unique cœur. Il semblerait donc que `valgrind` ne soit pas capable de profiler sur plusieurs cœurs, que sa machine virtuelle ne soit qu'un processeur mono-cœur. Ceci signifie que cela peut avoir un impact sur la justesse du profilage, puisque les threads se gênent mutuellement et ne peuvent pas profiter de l'intégralité de la machine.

Ce profilage sur une véritable application scientifique a également permis de sentir les limites de la sortie textuelle du profileur. Dans le cadre d'une application parallèle, le profileur affiche la pile d'exécution de chaque fonction (comme pour le séquentiel), pour chaque thread. Dans le cas des pilotes threads, où il n'y a que quatre threads, l'affichage est verbeux, mais encore exploitable. Dans le cadre de la simulation avec la centaine de threads, on est noyé sur l'information et cet affichage se révèle peu exploitable, en tout cas pour les piles d'exécutions. D'autant que, dans le cadre de notre simulation, il s'agit toujours des mêmes fonctions lancées sur plusieurs threads, on se retrouve donc avec une information doublonnée. Le compte rendu final, avec l'ensemble des coûts par fonction en revanche reste totalement exploitable, les coûts étant fusionnés entre les threads. Néanmoins, on sent clairement la limite des compteurs. Des compteurs plus précis permettraient d'extraire plus d'informations sur le profilage.

3.5.3 – Profilage à l'aide de C++11

On pourra également noter que C++11 autorise à faire de la programmation orientée aspect à un niveau similaire à celui fourni par `AspectC++`. En effet, il est possible de faire du *wrapping* de fonction pour rajouter des aspects. Dans C++11 ont été rajoutés les « *variadics templates* », c'est-à-dire des *templates* à arguments variables (par le type et par le nombre). Cela remplace, avec bien plus de puissance, les *variadic functions* du C99 (les fonctions à arguments variables). C'est ainsi qu'on pourrait imaginer un profileur fait en C++11. Le profileur aurait la forme décrite sur le Code 19.

```
template<typename FunctionType, typename… Args>
auto ProfilerAspect (FunctionType * Function,
                     const Args&… FunctionArgs) ->
                                        decltype(Function(FunctionArgs…))
{
…
}
```

Code 4-19 : Signature de la fonction de profilage pour un profileur en C++11

Le code 19 peut être vu en deux parties. La première partie est la déclaration stricte du *template*, qui permettra d'avoir le type des arguments pour tout le reste de l'appel. On a donc en premier lieu, le type de la fonction qui sera profilée, cela correspond globalement à son prototype. Puis, le reste du *template* est « *variadic* », il contiendra tous les types de tous les arguments passés à la fonction. La

seconde partie est donc le prototype de la fonction de profilage, qui s'appelle `ProfilerAspect()`. Les arguments suivants sont en premier lieu un pointeur vers la fonction dont le type est le premier du *template*, puis la liste « *variadic* » des arguments, comme pour le *template*. Le mot clé « `auto` » qui se substitue au type de retour habituel permet de laisser le compilateur, au moment de l'instanciation du *template* de choisir le bon type de retour, grâce au type de retour de la fonction, ce qui lui est précisé grâce au mot clé « `decltype` ».

On voit donc poindre le défaut de l'implémentation native avec C++11, il n'y a pas d'étape de tissage (« *weaving* »), il n'y a donc pas de points de coupure. Ainsi, en admettant que l'on ait une fonction qui fait l'addition de deux nombres, qui a le prototype du Code 20, alors, pour pouvoir la profiler, il faudrait l'appeler comme sur le Code 21. Ceci serait plutôt intrusif, mais la seule limite ici, serait l'implémentation, et non plus l'outil utilisé pour avoir l'aspect, puisque tout serait géré par le compilateur.

```
int Add(int a, int b);
```

Code 4-20 : Prototype d'une fonction pour ajouter deux entiers

```
Res = ProfilerAspect(&Add, 9, 11);
```

Code 4-21 : Utilisation de l'aspect pour le profilage sur la fonction `Add(9, 11)`

On peut donc constater sur le Code 21, que si l'utilisation du C++11 permet d'éviter un outil supplémentaire dans la chaîne de compilation, il introduit une lourdeur manifeste dans le code source, et moins de souplesse pour activer ou désactiver l'aspect. C'est pourquoi, nous n'avons pas plus poussé l'implémentation en C++11.

4 – Conclusions

Dans la première partie de ce chapitre, nous avons présenté les optimisations et les améliorations mises en œuvre dans la simulation `tomusim`. La mise en œuvre de toutes ces améliorations a permis d'obtenir un gain de performance de l'ordre de 415X en utilisant un nœud de calcul avec 32 cœurs physiques par rapport à l'application séquentielle initiale. Il s'agit du plus grand gain de performance rencontré sur l'intégralité de cette thèse. Si ce gain de performance a ouvert des perspectives permettant de rajouter des fonctionnalités dans la simulation originale, celle-ci a

néanmoins été abandonnée au profit d'une nouvelle simulation `tmvg4sim`, que nous avons décrite au chapitre précèdent.

Dans ce chapitre, nous avons également mis en pratique la proposition du chapitre précèdent visant à créer un profileur en programmation orientée aspect pour le C++. A l'origine prévu pour profiler `tomusim`, nous l'avons finalement testé sur la simulation qui sera détaillée au chapitre suivant. Malgré une première estimation qui semblait bonne, notre profileur accuse d'un *overhead* important, de l'ordre de de 100X, tandis que `valgrind` inflige un *overhead* plus limité de seulement 25X. Cependant, nous avons pu constater que notre profileur semble mieux gérer le parallélisme que `valgrind`, n'écrasant pas la simulation parallèle sur un unique cœur, et fournissant plus d'informations sur les threads que `valgrind` (qui n'en fournit pas).

Chapitre 5 : Développement d'une simulation reproductible, statistiquement correcte et fortement parallèle

1 – Introduction

Dans le chapitre de propositions, nous avions proposé plusieurs contraintes de développement et de conception pour avoir une application parallèle qui soit reproductible, statistiquement correcte et qui puisse passer à l'échelle. Nous avions également proposé une méthode permettant de vérifier si oui ou non l'application ainsi créée répond à ces critères.

L'application de démonstration de ces principes est une simulation de Monte Carlo, mise au point pour simuler la propagation des muons à travers un simple bloc de matière. Dans un premier temps, nous expliquerons en détail l'implémentation de la simulation, puis, nous nous intéresserons à la (non-)reproductibilité numérique de la simulation. Enfin, nous nous pencherons sur les performances de la simulation, à la fois sur CPU « classique » de type multi-cœurs et sur Xeon Phi.

2 – Développement de la simulation

2.1 – Modèle physique

Le modèle physique derrière notre simulation est très simple. Ici, il n'est pas question de structure complexe telle que le Puy-de-Dôme. Notre simulation va générer des muons à 1,999 km de la cible. La cible est un parallélépipède de matière de 2 km de long et de 5 m de hauteur et de largeur. La cible et le détecteur sont séparés par 1 m. On a donc une distance de 4 km entre la génération de la particule et le détecteur. Cette disposition est illustrée sur la figure 1.

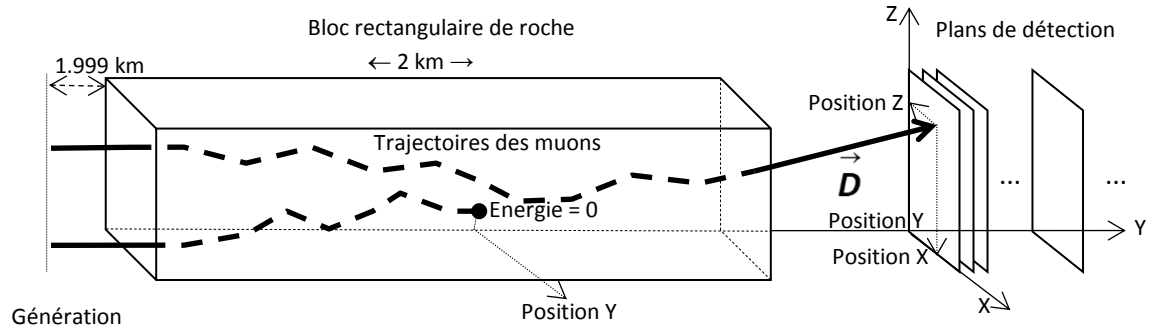

Figure 5-1 : Le modèle physique

Comme on peut le constater sur la figure 1, le détecteur est constitué de plusieurs plans de détections de taille 1 m^2, qui permettent notamment d'estimer la trajectoire du muon et de corréler les évènements. Dans le cadre de nos simulations de la propagation du détecteur, nous laissons toujours la simulation du détecteur à un autre programme. En effet, l'objectif est de pouvoir séparer les deux simulations et les rendre totalement indépendantes. Ainsi, on peut avec l'une des simulations générer un ensemble de particules propagées et avec l'autre, simuler leur passage à travers un détecteur. Dès lors, on peut simuler différentes configurations de détecteur tout en conservant les mêmes particules et observer les différences dans les résultats du détecteur.

Dans notre cas, nous nous occuperons uniquement de la première simulation : la propagation des muons à travers le bloc de roche. Le bloc de roche est considéré comme faite de « roche standard » selon la définition du PDG (*Particle Data Group*)[31] (Lohmann, Kopp, & Voss, 1985). Les muons sont caractérisés par cinq propriétés : énergie, temps propre (i.e., son temps de propagation), longueur de propagation, position et direction. Ces données sont initialisées aléatoirement à la création du muon. La position du muon est donc dans une surface de génération à quatre kilomètres du premier plan du détecteur. Cette surface de génération est un carré de 20 m^2. La propagation des muons est gérée par une bibliothèque, écrite en C, dédiée. Elle s'appuie sur des méthodes semi-analytiques pour condenser tous les processus physiques sous-jacents, tout en étant optimisée pour l'utilisation du CPU. De façon schématique, la propagation du muon est une séquence d'étapes discrétisées durant lesquelles la direction du muon est aléatoirement défléchie et son énergie perdue au profit du medium de propagation. La propagation s'achève lorsque le muon a perdu toute son énergie et/ou lorsqu'il a atteint le premier plan de détection, dans notre cas quand sa coordonnée Y vaut 0 (cf. figure 1).

[31] http://pdg.lbl.gov/2014/AtomicNuclearProperties/HTML/standard_rock.html

2.2 – Implémentation de la simulation

La simulation elle-même a été développée en C++ avec un minimum de dépendances, pour faciliter son portage vers d'autres architectures (notamment, le Xeon Phi). Ainsi, elle ne s'appuie que sur les fonctions de la bibliothèque standard C++ et sur sa bibliothèque interne en C, qui elle n'a aucune dépendance. Parce que c'est une application scientifique, tous les nos calculs en virgule flottante sont réalisés en double précision. Afin d'avoir plusieurs flux pseudo-aléatoires disponibles facilement dans la simulation, nous avons utilisé le paquet « `RngStream` » de L'Ecuyer (L'Ecuyer et al., 2002) qui s'appuie sur le générateur de nombres pseudo-aléatoires MRG2k3a (L'Ecuyer, 1999). Celui-ci, disponible sous la forme d'une classe C++ dispose d'une propriété très intéressante pour notre simulation : à chaque fois qu'il est instancié, il fournit un flux pseudo-aléatoire indépendant différent. Et les flux sont reproductibles et forment une suite : les flux se suivent dans le même ordre une fois le premier initialisé. Pour notre simulation, il suffit donc d'initialiser de façon fixe l'état du premier générateur, ensuite, nous parcourrons toujours les mêmes flux, dans le même ordre. Ceci est critique pour la reproductibilité numérique de la simulation.

De la même façon, nous avons suivi les contraintes explicitées dans le chapitre proposition : chaque muon aura son propre thread, et donc son propre flux pseudo-aléatoire. Ceci simplifie grandement l'implémentation : il suffit d'instancier le générateur dans le thread. On a alors des muons qui sont caractérisés par leur flux, puisque c'est à partir de celui-ci qu'est généré le muon. Quand la propagation du muon est finie, le thread lui-même est terminé, de même que le flux. Ce modèle suit parfaitement les recommandations énoncées et nous assure la reproductibilité. En effet, quel que soit le paradigme avec lequel sera lancé la simulation, séquentiel (i.e., un thread à la fois), parallèle (i.e., X threads à la fois), distribué (i.e., X instances séquentielles à la fois), la simulation se résume uniquement à l'exécution de threads de calcul indépendants. Nous aurons toujours les mêmes threads, leur ordre d'exécution diffère, mais cela n'a aucune importance sur les résultats finaux, leur indépendance est assurée grâce au seul statut initial du générateur de nombres pseudo-aléatoires (maître de la « vie » du muon).

Pour nous assurer la bonne gestion des flux stochastiques et des threads, nous avons développé une « *thread factory* ». Cette usine à threads remplit plusieurs fonctions. La première et la plus importante est de s'assurer qu'un seul thread sera créé à la fois. Si cela peut paraître contre-productif vis-à-vis des objectifs de la simulation, mais cela est rendu nécessaire par l'usage du générateur de (L'Ecuyer et al., 2002). En effet, comme précisé, celui-ci utilise un « générateur » de statut pour chaque initialisation du générateur. De fait, celui-ci est mis à jour à chaque instanciation

du générateur : quand on instancie le générateur, le statut pour le prochain est généré en même temps. Il devient donc critique de protéger l'instanciation du générateur, qui a lieu à la création du thread. Si on ne rend pas cette section critique, c'est-à-dire si on ne s'assure pas qu'un seul thread à la fois peut être créé, on se retrouve avec des situations de compétition entre threads pour l'accès au statut initial et pour la modification dudit statut. On pourrait se retrouver alors avec soit un statut en double, donc, deux fois le même muon, avec la même existence alors que ce biais statistique est évitable, ou alors un statut « intermédiaire » qui aurait un bout de l'ancien statut et un bout du nouveau. Ce type d'approche amènerait sur un flux stochastique global dont on ignore tout des propriétés statistiques, et qui serait de plus totalement non reproductible. D'autres approches pour obtenir des flux stochastiques indépendants existent comme nous vous l'avons montré précédemment (Hill et al., 2013), nous avons retenu celle proposée par (L'Ecuyer et al., 2002) dans `RngStream` et donc implémenté un mécanisme fiable pour l'association de flux indépendants à chaque thread.

Ainsi, quand la création du thread est initiée par l'usine à threads, il y a d'abord une entrée dans une section critique, et c'est à la fonction lancée dans le thread, c'est-à-dire dans notre cas, celle de propagation des muons, de signaler que son initialisation s'est bien déroulée et que d'autres threads peuvent être lancés. Dans notre cas, la fonction du thread signale dès qu'elle est lancée que l'initialisation est terminée, puisqu'il s'agit uniquement de l'initialisation du générateur sur la pile, ce qui est fait dès le lancement de la fonction. On se retrouve donc avec une pénalité très réduite pour le lancement séquentiel des threads. Bien évidemment, une fois le lancement du thread fait, plusieurs threads peuvent s'exécuter en parallèle pour obtenir une simulation parallèle.

Cette usine à threads est également là pour fournir une autre fonctionnalité de la simulation. Comme précisé dans le chapitre de propositions, lors de l'implémentation séquentielle, on doit penser la simulation parallèle pour que le passage à l'échelle puisse se faire facilement. Cette usine à threads permet de répondre à cette problématique très facilement. En effet, elle permet elle-même de limiter le nombre de threads qui peuvent s'exécuter en parallèle dans la simulation. Typiquement, si l'on veut une simulation séquentielle, il suffit de lui astreindre le nombre de threads maximum à un. Le cœur de la simulation, pour autant, reste le même. En effet, comme nous l'avons explicité, nous avons retenu une approche proposant un thread par particule, donc, la boucle des évènements se résume à être une boucle de lancement de threads. Qu'on ait un ou trente threads en parallèle, la boucle ne change pas. Seule l'usine à threads gère la différence. Pour se faire, elle se contente de mettre l'appelant en attente lorsqu'aucun thread n'est disponible pour le lancement : quand la boucle des évènements tente de lancer un nouveau thread pour un nouveau muon, mais

que le maximum de threads en parallèle est déjà atteint (pour s'ajuster aux capacités des processeurs disponibles), alors elle est mise en attente jusqu'à la prochaine disponibilité de lancer un nouveau thread (i.e., le décès d'un autre thread).

Cette implémentation est, dans notre cas spécifique, particulièrement performante : en effet, outre la section critique (limitée) en début de lancement de thread, le thread n'a plus besoin d'interagir avec les autres éléments de la simulation, si ce n'est des structures de données en lecture-seule qui sont immutables, donc invariants dans le temps. Il n'y a donc pas besoin d'un quelconque verrou pour y accéder. Une fois le thread lancé, il s'exécute sans interruption (volontaire) sur son cœur de calcul jusqu'à la fin de la propagation de la particule.

On notera également que conformément à ce qui a été dit dans le chapitre propositions, même si la simulation est séquentielle, on retrouve quand même un thread supplémentaire, qui est le thread d'écriture. Les entrées/sorties sont toujours découplées de l'exécution, que la simulation soit parallèle ou séquentielle.

Afin de faciliter l'exécution de plusieurs instances de la simulation pour pouvoir la distribuer sur plusieurs nœuds de calcul, en plus du nombre d'évènements, la simulation peut prendre en argument l'évènement auquel commencer. Typiquement si on veut simuler deux millions d'évènements sur deux machines, on pourra lancer une première instance pour qu'elle simule le premier million d'évènements (de 0 à 999 999) et la seconde instance simule le second million d'évènements (de 1 000 000 à 1 999 999). Ceci permet de faciliter le déploiement de la simulation sur différentes machines sans se poser la question des flux stochastiques et de la gestion des initialisations des générateurs. Ceci a été implémenté simplement en faisant, en début de programme, « sauter » les évènements à la simulation : le générateur correspondant à l'évènement est généré (pour avancer dans le statut global) puis immédiatement supprimé pour passer à l'évènement suivant, jusqu'à ce qu'on arrive aux évènements que l'on souhaite réellement simuler. Cette opération est faite ainsi, parce que le générateur a un mécanisme de « *jump-ahead* » suffisamment rapide pour que cela passe inaperçu au démarrage (L'Ecuyer, 2010). Il suffit donc à l'utilisateur d'avoir proprement découpé son plan d'expérience pour pouvoir distribuer la simulation sur plusieurs nœuds.

Enfin, par mesure de précaution, à l'issue de l'implémentation, nous avons passé notre simulation dans `valgrind` (Nethercote & Seward, 2003) avec les outils `drd` et `helgrind` (Valgrind-project, 2007). Ceux-ci permettent notamment de vérifier le bon usage des threads et des zones mémoires dans le cas des applications parallèles. L'objectif étant de détecter la mauvaise utilisation des

verrous (mutex, sémaphores, etc.) et de signaler les risques d'inter-blocage mais aussi de détecter les situations de compétitions entre threads. Quel que soit le paradigme d'exécution choisi, notre simulation est passée sans avertissement ni erreur dans ces deux outils de `valgrind`. Ceci nous a permis une validation statistique de son bon fonctionnement parallèle, et de s'assurer que nous n'avons pas de sources d'aléa (et de non reproductibilité) non maîtrisées.

2.3 – Modifications parallèles pour le Xeon Phi

Les tests de notre simulation pour Xeon Phi (voir plus bas), nous ont permis de constater que le Xeon Phi était incapable de monter à pleine charge avec le modèle de parallélisme correspondant à la proposition d'un thread par évènement. En effet, le Xeon Phi n'était capable de lancer qu'un thread par cœur physique (donc qu'un thread sur les quatre possibles, un cœur physique de Xeon Phi disposant de quatre cœurs logiques – ou threads matériels). Même en tentant de lancer 240 threads sur le Phi, seulement 25% du Phi était exploité et quatre threads de la simulation étaient écrasés sur un seul thread matériel du Xeon Phi. Ce comportement de la simulation, et plus particulièrement de `pthread` nous rappelle un problème observé dans (Innocenti, Silvani, Muzy, & Hill, 2009; Innocenti, 2004), où les `pthreads` n'arrivaient pas à migrer sur les cœurs logiques. Clairement, les performances n'étaient pas au rendez-vous. Nous avons donc dû changer notre modèle de parallélisme vers un modèle plus classique (plus proche de (Hill, 2015)), qui délègue une partie de la boucle des évènements dans chaque thread. Ainsi, la simulation distribue les évènements à réaliser entre les différents threads, et chaque thread gère plusieurs évènements. A titre d'exemple, pour une simulation d'un million d'évènements, pour l'ancien modèle, cela signifie 1 millions de threads d'un évènement. Avec ce nouveau modèle de parallélisme, cela signifie qu'il n'y aura que 240 threads (pour Xeon Phi) d'environ 4 166 évènements. Cela signifie qu'il faut gérer un peu différemment les générateurs de nombres pseudo-aléatoires. Ici, on déplace la complexité non plus au niveau de la création du thread, mais au niveau de la boucle des évènements du thread directement. C'est elle qui doit gérer la section critique à chaque changement d'évènement, puisque pour, pour conserver la reproductibilité, chaque évènement dispose toujours de son propre générateur, instancié avant le muon lui-même.

Au niveau des performances, nous n'avons pas noté de changement entre les deux modèles de parallélisme. Nous avons donc fait le choix de conserver les deux ; le choix étant fait à la compilation. On garde donc le modèle qui implémente rigoureusement les propositions et qui fonctionne parfaitement sur CPU, et on propose un modèle alternatif plus hybride qui fonctionne sur Xeon Phi.

Par analogie au monde de la grille de calcul (Sfiligoi, Quinn, Green, & Thain, 2008), nous avons nommé ce modèle de threading plus classique « Job pilote de threads » (en anglais « pilot thread »).

2.4 – Stockage des résultats

Afin de stocker les résultats, nous avons fait un choix important au niveau de la représentation des données, afin de faciliter leur gestion, et nous permette encore une fois, une reproductibilité numérique plus aisée. Plutôt que d'utiliser un format complexe qui impliquerait un retraitement des données pour fusionner les fichiers de sortie en cas de distribution de la simulation sur plusieurs nœuds de calcul, par exemple, nous avons décidé d'utiliser un fichier binaire très simple dans lequel nous nous contentons de mettre les données des évènements les uns après les autres, en agençant les données pour un évènement selon une structure donnée. Ainsi, quand il s'agit de regrouper les données de plusieurs instances de la simulation, il suffit de mettre les fichiers « bout à bout » et on obtient les résultats pour l'ensemble de la simulation.

Par ailleurs, ce format de stockage de données est totalement compatible avec le modèle utilisé pour les entrées/sorties. Nous avons mis en place un seul thread d'écriture : dès que notre thread d'écriture reçoit un évènement, il met en forme la structure de données et l'écrit dans le fichier. Etant donné qu'il n'y a qu'un thread d'écriture, le fichier est géré de façon séquentielle, et on a la garantie que les évènements sont proprement écrits. Par ailleurs, étant donné qu'ils sont écrits dès qu'ils sont disponibles, il n'y a pas de consommation mémoire liée simplement à la conservation des résultats en vue de leur mise en forme finale avant écriture. Les évènements étant stockés en brut, cela permet également de les retraiter comme on le souhaite par ailleurs, sous réserve d'écrire le bon programme pour les lire. Ce choix a été fait pour des raisons de performances : les sorties du programme (et les temps d'attente induits) ne grèvent pas les calculs. Que le résultat soit écrit ou non, la simulation peut continuer les calculs, les écritures étaient faites les unes à la suite des autres, quand le stockage est disponible pour une écriture. Cela peut éventuellement poser un problème de performance avec de nombreux threads en parallèle (un thread d'écriture pour X threads de calcul, par exemple), cependant, cela signifie juste que quand tous les calculs seront terminés, il faudra juste attendre sur ce thread d'écriture. Cela ne générait pas autant que X threads qui tentent d'accéder au stockage, en même temps, sur le même fichier pour écrire chacun leurs données (et les problèmes de cohérence que cela causerait).

Nos fichiers de sortie contiennent donc la direction finale du muon, la position finale du muon, son énergie finale, son temps propre ainsi que sa longueur de propagation. En plus de ces six attributs, nous en avons rajouté un septième qui correspond à l'identité du muon : le statut initial du générateur. Notre générateur étant reproductible, et ayant une bijection entre muon et statut initial du générateur, ceci nous permet d'identifier à chaque fois, de façon certaine notre muon, afin de pouvoir comparer les résultats et éventuellement, leurs divergences.

3 – Reproductibilité

3.1 – Reproductibilité sur le même matériel

Pour rappel, dans notre cas, la reproductibilité signifie que quel que soit le paradigme d'exécution choisi pour notre simulation, quelle que soit l'architecture utilisée pour l'exécution, si nous générons dix mille évènements, dans les mêmes conditions, nous devons toujours avec les mêmes dix milles évènements, et si possible avec une reproductibilité « *bit-for-bit* ». L'ordre des évènements importe peu.

La première étape est donc de vérifier que nous avons la reproductibilité « *run-to-run* », c'est-à-dire que deux exécutions dans les mêmes conditions donnent les mêmes résultats. Si cela peut sembler évident, ce ne l'est pas (Rosenquist, 2012). Une situation de compétition, par exemple, peut l'empêcher. Dans notre cas, nous avons pu vérifier sur notre machine de tests, équipée de deux processeurs E5-2687W[32], que cette reproductibilité était vérifiée, et ce au bit près : toutes les valeurs résultat sont strictement identiques, bit-à-bit.

Dans un second temps, nous avons comparé, toujours sur la même machine, la même simulation avec les trois paradigmes d'exécutions différents : séquentiellement, parallélisé avec 32 cœurs et distribué en 32 instances (chacune étant utilisée avec le modèle séquentiel). Dans les deux premiers cas, un seul fichier de sortie a été généré. Dans le dernier cas, nous avons simplement agrégé les 32 fichiers de sortie en un à l'issue de l'exécution des instances. Nous avons constaté ici, à nouveau, une parfaite reproductibilité numérique, au bit près. Seul l'ordre des évènements dans les fichiers change, mais cela n'a aucune espèce d'importance dans notre cas. Et, étant donné que les paradigmes d'exécution changeaient, nous nous doutions que l'ordre changerait. Néanmoins, seul le

[32] http://ark.intel.com/fr/products/64582/Intel-Xeon-Processor-E5-2687W-20M-Cache-3_10-GHz-8_00-GTs-Intel-QPI

fait que chaque évènement ait suivi la même vie et donne le même résultat à la fin, strictement identique bit-à-bit nous importe.

3.2 – Reproductibilité Xeon CPU / Xeon Phi

Nous nous sommes ensuite penchés sur la reproductibilité dans le cas d'un changement de matériel, en comparant les résultats entre Xeon CPU et Xeon Phi. Nous avons toujours lancé la simulation de la même façon sur les deux architectures : séquentielle, sans sauter d'évènements, puis, nous avons comparé les résultats ainsi obtenus.

3.2.1 – Compilation simple

Dans un premier temps, nous avons simplement étudié deux compilations naïves de la simulation. Une pour nos processeurs x86 classiques E5-2687W et une pour le Xeon Phi 5110P. Dans les deux cas, nous avons utilisé le compilateur C d'Intel avec les options de compilation « `-O2 -g -W -Wall -Wextra` ». Pour le Xeon Phi, nous avons rajouté l'option « `-mmic` » qui permet de compiler pour Xeon Phi et pour le CPU, nous avons rajouté l'option « `-xHost` » qui permet de coller la compilation à la machine, en utilisant le plus haut jeu d'instructions disponible. Pour l'exécution, nous avons utilisé les threads « pilotes », avec une exécution séquentielle. L'objectif étant d'étudier la validité des résultats et non les performances, dans un premier temps.

Nous savions que cette tentative ne conduirait pas à la reproductibilité. Mais cela permet d'évaluer la déviation dans les résultats quand la compilation est laissée libre. Nous nous sommes donc limités à la simulation de 1000 évènements, en sachant que nous perdrions déjà la reproductibilité sur cet échantillon. En regardant ces 1000 évènements, nous avons immédiatement constaté des divergences très importantes. Des différences en énergie (finale) allant jusqu'à 0.18 GeV, mais aussi et surtout, pour la position finale des différences allant jusqu'à 0,3 m. Si l'énergie initiale de la particule est comprise en 5 GeV et 10 TeV, son énergie finale est comprise entre 0.15 GeV et 5 TeV (voire nulle, si elle n'atteint même pas le détecteur). Une différence de 0.18 GeV n'est donc plus du tout acceptable. Pour la position finale, on rappelle que le détecteur a des plans dont la dimension est d'un mètre sur un mètre. Une imprécision de 0,3 m sur la position finale signifie une imprécision de 30% sur une dimension du plan ! C'est extrêmement important et pourrait conduire une particule

à passer à travers le détecteur dans un cas, et à côté dans l'autre. Pire, le détecteur a une détection spatiale de l'ordre de 1 cm. Une imprécision de l'ordre 30 cm (i.e., 30 fois plus !) est donc un échec clair et net de la reproductibilité de la simulation.

3.2.2 – Compilation avec « `-fp-model precise` »

Ces différences n'étant clairement pas acceptables, nous avons renforcé la « justesse » de la simulation en utilisant l'option de compilation « `-fp-model precise` » lors de la compilation sur les deux architectures. Ce paramètre est fortement recommandé par Intel, puisqu'il aide fortement à la reproductibilité. Néanmoins, comme nous avons pu le constater (et comme nous allons le montrer ci-dessous), il n'est pas suffisant. Pour ce test, nous avons travaillé sur 5 000 évènements, avec le même modèle d'exécution que précédemment.

D'emblée, nous avons pu constater un élément intéressant : nous avons des valeurs qui sont reproductibles bit-à-bit. Chose totalement inimaginable dans le cas précédent, néanmoins, de tels cas sont rares, et ne concernent que quelques valeurs pour des évènements, les autres valeurs ne sont pas reproductibles bit-à-bit. Nous nous sommes alors intéressés à la différence relative des valeurs telle que définie dans le chapitre de propositions. Nous avons comparé chacune des valeurs des 5 000 résultats (9 valeurs par résultat), ce qui nous donne, en tout un échantillon de 45000 valeurs à manipuler. Néanmoins, en raison de ce qui a été évoqué au chapitre des propositions, en raison de la nature de la valeur « Position Y » qui correspond à la coordonnée de la particule atteignant le plan de détection en Y = 0, nous ignorerons totalement cette valeur pour ne pas fausser notre étude : nous travaillerons donc sur un échantillon réduit de 40 000 valeurs. Nous avons noté les différences relatives de ces résultats dans le Tableau 1. On a donc un total de 5 000 valeurs par colonnes.

Tableau 5-1 : Différences relatives des valeurs entre Xeon Phi et Xeon CPU

	Position X	Position Z	Direction X	Direction Y	Direction Z	Distance	Energie	Temps
0	0	0	0	2	0	4641	3303	4372
10^{-16}	0	0	0	29	0	333	1310	91
10^{-15}	0	0	0	333	0	20	343	138
10^{-14}	4	2	0	4326	0	6	43	43
10^{-13}	17	17	6	149	12	0	1	94
10^{-12}	164	184	104	26	83	0	0	247
10^{-11}	1065	1013	696	1	676	0	0	15

10^{-10}	2578	2583	2555	4	2600	0	0	0
10^{-9}	1041	1061	1360	4	1324	0	0	0
10^{-8}	113	130	157	12	162	0	0	0
10^{-7}	14	9	37	26	50	0	0	0
10^{-6}	4	1	48	55	59	0	0	0
10^{-5}	0	0	32	28	29	0	0	0
10^{-4}	0	0	5	5	4	0	0	0
10^{-3}	0	0	0	0	1	0	0	0
Médiane	1.60E-10	1.63E-10	2.54E-10	1.98E-14	2.48E-10	0	0	0
Moyenne	1.65E-09	1.09E-09	2.56E-07	1.79E-07	3.34E-07	3.14E-17	3.12E-16	1.11E-13
Maximum	8,47E-07	7,90E-07	2,99E-04	1,03E-04	6,27E-04	4,31E-16	4,63E-15	5,57E-13

En première observation, on peut constater dans le Tableau 1, que pour certaines valeurs distance (longueur de propagation), énergie et temps (propre), on a quasiment tout le temps une reproductibilité bit-à-bit (ligne « 0 »), à tel point que la médiane pour ces évènements est en 0. Et quand il n'y a pas reproductibilité bit-à-bit, la différence reste faible. En revanche, pour le reste des valeurs, position et direction, la reproductibilité bit-à-bit devient rarissime : atteinte uniquement deux fois sur 25 000 valeurs (i.e., 0,08‰) ! La ligne 10^{-n} donne ainsi le nombre de valeurs dont la différence relative entre les deux architectures est entre 0.5 x 10^{-n} et 5 x 10^{-n}. On peut dès lors constater que les différences peuvent s'étaler sur un très large spectre. On constate même que pour ces valeurs, dans tous les cas, la moyenne est d'un ordre de grandeur (minimum !) supérieur à la médiane, ce qui implique un étalement des différences dans les « queues » de leur distribution, y compris dans les grands ordres de grandeur de différence. A titre d'exemple, considérons la colonne « Direction Y ». Pour cette valeur, la majorité des différences relatives (4 326 valeurs parmi les 5 000) est de l'ordre de 10^{-14}, d'où une médiane du même ordre de grandeur. Mais pourtant, 88 valeurs ont une différence relative supérieure à 10^{-6}, allant même jusqu'à une différence relative de l'ordre de 10^{-4}.

Nous nous sommes alors intéressés aux nombres de bits (dans la représentation IEEE 754) qui pourraient différer dans les résultats entre les deux architectures, en raison de l'absence de reproductibilité bit-à-bit entre les deux architectures. Toutes nos valeurs sont calculées en double précision, c'est-à-dire qu'elles sont représentées sur 64 bits. D'après la norme IEEE, les 64 bits sont utilisés de la façon suivante : 1 bit de signe (le bit de poids fort), s, les 11 bits suivants permettent de représenter l'entier correspondant à l'exposant du nombre, e. Enfin, les 52 bits restants représentent la mantisse du nombre, m. On notera que nos nombres sont des nombres normalisés (Goldberg, 1991), c'est-à-dire que m est $1 \leq m \leq 2$ car le 53ème bit (caché) de la mantisse vaut toujours 1, d'où son absence de représentation dans la mantisse, d'ailleurs. Ainsi, un nombre est

calculé de la façon suivante : $(-1)^s \times m \times 2^e$. Pour l'étude des changements, nous ne nous sommes intéressés qu'à la mantisse. En effet, même si des changements de signe et des changements d'ordre de grandeur pour les valeurs sont théoriquement possibles, ils devraient être suffisamment rares. La donnée fluctuante est donc concentrée dans la mantisse.

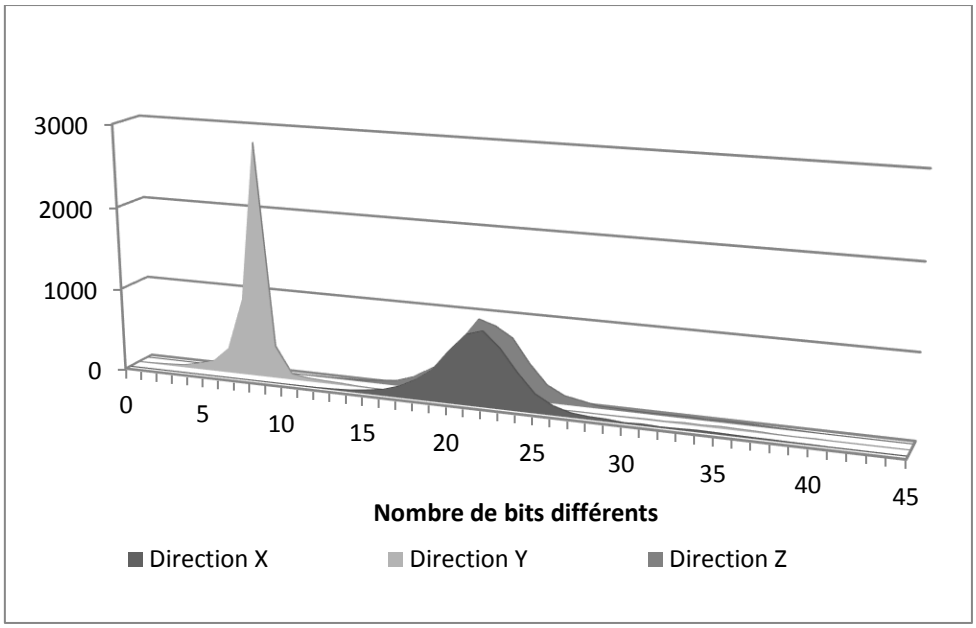

Figure 5-2 : Nombre de bits potentiellement différents entre le Phi et le CPU

La Figure 2 ci-dessus montre le nombre de bits qui peuvent potentiellement différer sur les trois composantes de la direction, entre les résultats du Xeon Phi et du CPU. La différence maximale entre les deux architectures est de 40 bits pour la direction Y : les 13 bits restants correspondent à la représentation de la différence relative maximale (1,03 x 10^{-4}). Le nombre n de ces bits restants peut être obtenu grâce au calcul : étant donné la différence relative Δ, $n = floor(-log_2 \Delta)$. On obtient d'ailleurs le nombre de bits potentiellement différents grâce à la soustraction $53 - n$: la mantisse est sur 53 bits !

En regardant attentivement le tableau 1, on peut émettre l'hypothèse que compte tenu de la répartition des ordres de grandeurs des différences relatives, il pourrait exister des évènements pour lesquelles l'ordre de différence pourrait atteindre un seuil critique, en raison de la longue dispersion vers les ordres de grandeurs importants (cf. Direction Y). En effet, nous avons pu constater, après avoir classé les différences par ordre croissant pour la Direction Y, que la différence maximale (1,03 x 10^{-4}) est approximativement 2,5 fois plus grande que la 5ème plus grande différence

(99,9% fractile = 3,89 x 10^{-5}), mais également 41,5 fois plus grande que la 50ème plus grande différence (99% fractile = 2,49 x 10^{-6}). On peut donc légitimement se demander si ces valeurs ne vont pas venir tellement grandes que la reproductibilité de la simulation pourrait être remise en question. Afin de répondre à cette question, nous avons quelque peu modifié la simulation. En effet, nous avons grossi le problème en conservant le même modèle. De fait, les particules avaient X fois plus d'énergie initiale, pour parcourir une distance X fois plus grande dans laquelle se trouve un parallélépipède deux fois plus long. Les autres dimensions, elles restant inchangées. Le facteur multiplicatif X prenant pour valeurs 2, 4, 8, 16 et enfin 32. La valeur maximale de X étant limitée par la validité du modèle théorique utilisé. Les résultats que nous avons obtenus de la sorte ne nous ont pas permis d'établir qu'il y a une augmentation significative (en rapport avec l'augmentation de la taille du problème) de la différence relative entre les valeurs des résultats. Cependant, cela a permis de mettre en évidence que dans certains cas, nous atteignons des valeurs où la différence relative devient totalement critique, avec plus de 60% de différence ! Inutile de dire que dans ces cas-là, toute reproductibilité est perdue.

3.2.3 – Une compilation encore plus rigoureuse

Nous nous sommes donc tournés vers la compilation la plus rigoureuse, préconisée par Intel dans les cas où la reproductibilité importe le plus. Néanmoins, Intel ne promet toujours pas qu'il y ait toujours une reproductibilité bit-à-bit (Corden & Kreitzer, 2014). Nous avons donc recompilé l'ensemble de nos simulations pour Phi et pour CPU avec les options de compilations suivantes supplémentaires : « `-fp-model precise -fp-model source -fimf-precision=high` ». Pour le Xeon Phi, nous avons également désactivé la FMA (*Fused-Multiply-Add*) matérielle, comme préconisé par Intel, grâce à l'option de compilation « `-no-fma` ». Suivant le protocole défini précédemment, nous avons à nouveau généré 5 000 évènements, considéré les 40 000 valeurs des résultats (nous ignorons toujours la position Y). En ce qui concerne l'énergie, le temps propre et la longueur de propagation, il y a pour chaque évènement reproductibilité bit-à-bit. Les différences relatives pour les autres valeurs ont été classées dans le tableau 2.

Tableau 5-2 : Différences relatives des valeurs et bits modifiés entre Xeon Phi et Xeon CPU

Différence ↓ \ Valeur →	Position X	Position Z	Direction X	Direction Y	Direction Z
0 bit : reproductibilité bit-à-bit	4922	4934	4896	4975	4913
1 bit : 1.11E-16 ≤ Δ < 2.22E-16	25	21	14	5	18
2 bits : 2.22E-16 ≤ Δ < 4.44E-16	21	18	52	4	31
3 bits : 4.44E-16 ≤ Δ < 8.88E-16	15	12	23	6	12
4 bits : 8.88E-16 ≤ Δ < 1.78E-15	10	7	5	4	10
≥ 5 bits : 1.78E-15 ≤ Δ < 2.25E-11	7	8	10	6	16

En première observation du tableau 2, on peut constater que dans la majorité des cas, nous avons une reproductibilité bit-à-bit : 24 640 valeurs sur 25 000 (plus de 98% des valeurs sont reproductibles bit-à-bit). Si l'on considère toutes les valeurs, sur ces 5 000 évènements, on a 39 460 valeurs reproductibles bit-à-bit sur les 40 000, c'est-à-dire plus de 99% ! Nous nous sommes alors posé la question suivante : étant donné qu'une valeur pour un résultat R n'est pas reproductible bit-à-bit, qu'elle est la probabilité que la reproductibilité soit perdue pour d'autres résultats R', R'', etc. ?

Tableau 5-3 : Probabilités conditionnelles de la non-reproductibilité bit-à-bit

	Position X	Position Z	Direction X	Direction Y	Direction Z
Position X	1.0000	0.0758	0.5769	0.2800	0.0805
Position Z	0.0641	1.0000	0.0865	0.3600	0.4713
Direction X	0.7692	0.1364	1.0000	0.9200	0.2644
Direction Y	0.0897	0.1364	0.2212	1.0000	0.2529
Direction Z	0.0897	0.6212	0.2212	0.8800	1.0000
	0.0156	0.0132	0.0208	0.0050	0.0174

Le tableau 3 est conçu pour répondre à cette question, et est construit de la sorte : la dernière ligne du tableau contient la division du nombre de valeurs non reproductibles par 5 000. C'est une estimation de la probabilité que le résultat Ri ne soit pas reproductible. Par exemple pour la Position X, on a 5 000 – 4 922 = 78 valeurs non reproductibles bit-à-bit, d'où une probabilité de non-reproductibilité de 78 / 5 000 = 0,0156 soit 1,56 chances sur 100. Pour chaque colonne j, qui représente un résultat j, la cellule i, qui présente un résultat i, donne la probabilité conditionnelle

que la valeur ne soit pas reproductible bit-à-bit pour Ri, sachant qu'elle ne l'est pas pour Rj. A titre d'exemple, la probabilité conditionnelle que la valeur Direction X pour un résultat ne soit pas reproductible bit-à-bit, sachant que la valeur Position X ne l'est pas est de 0,7692. Cela signifie que parmi les 78 résultats ayant une valeur non reproductibles bit-à-bit pour la Position X, 0,7692 x 78 ≈ 60 d'entre eux auront une valeur non reproductible bit-à-bit pour la valeur Direction X. Le fait que cette probabilité conditionnelle (0,7692) soit bien plus élevée que la probabilité d'un résultat de ne pas être reproductible bit-à-bit pour la valeur Direction X (0,208) montre une forte dépendance entre les deux valeurs : s'ils étaient indépendants, la probabilité conditionnelle serait le plus proche possible de 0,208 et dès lors, seulement quelques résultats (1 ou 2) auraient des valeurs de Position X et de Direction X non reproductibles. De la même façon, dans le tableau 3, on peut constater une forte dépendance entre la (non reproductibilité bit-à-bit de la) Position Z et la Direction Z. On peut également remarquer si Direction Y n'est pas reproductible pour un résultat donné, alors les probabilités que ni Direction X et ni Direction Z ne soient reproductibles bit-à-bit sont élevées. De façon générale, les probabilités conditionnelles sont bien plus élevées que les probabilités générales, ce qui signifie que, d'un point de vue de la reproductibilité, ces valeurs ne sont pas indépendantes, et fournissent donc une information redondante. D'un point de vu algorithmique, cette interdépendance s'explique facilement par le fait que le vecteur direction est utilisé pour établir la nouvelle position à chaque pas de la simulation. Les deux sont donc clairement liés algorithmiquement.

Dans certains cas rares, la différence relative entre deux valeurs du Phi et du CPU peut excéder 5 bits de différence. Dans le cas des 5 000 évènements que nous avons simulés, la différence maximale obtenue a été atteinte pour la Direction Z avec $2{,}25 \times 10^{-11}$ (ce qui fait une différence de 18 bits, ce qui veut dire qu'il en reste tout de même encore 34 identiques !).

Nous avons à nouveau testé la simulation en grossissant le problème avec les facteurs multiplicatifs donnés précédemment. Une fois encore, nous n'avons constaté aucune augmentation significative de la différence relative comparativement à l'augmentation de la taille du problème. Cependant, nous avons constaté que la reproductibilité bit-à-bit était éventuellement perdue pour l'énergie et le temps propre (contrairement au problème en taille originale). En effet, sur les 35 000 résultats obtenus, deux résultats présentaient des valeurs non reproductibles : 3 bits de différence sur l'énergie pour le premier résultat, 3 et 10 bits de différence respectivement pour l'énergie et le temps propre pour le second résultat. De plus, la différence maximale obtenue est de $2{,}22 \times 10^{-8}$; elle est obtenue sur la Position Z. Cette différence est particulièrement élevée par rapport à toutes

les autres obtenues, mais elle reste néanmoins inférieure à la précision maximale que l'on peut obtenir en faisant du calcul en virgule flottante en simple précision, qui est de l'ordre de 5,96 x 10^{-8}.

Ainsi, lorsque la précision est poussée au maximum et le compilateur fortement contraint dans ses optimisations et ses arrondis, on est capable d'obtenir une reproductibilité bit-à-bit complète (i.e., pour chaque valeur) quand on considère une précision de l'ordre de la simple précision en virgule flottante. Dans le cadre de notre simulation, étant donné le modèle physique simulé et le bruit de fond ambiant observé lors des mesures réelles, cette reproductibilité en simple précision est totalement acceptable : notre simulation est donc reproductible.

Néanmoins, on constate que cette reproductibilité a un coût. En effet, entre cette compilation et la précédente (celle du 3.2.2), le temps d'exécution de la même simulation a augmenté de 20%, passant de 49h environs à presque 60h de calcul. Ce coût est nécessaire pour s'assurer de la bonne reproductibilité de la simulation. En effet, comme montré précédemment, sans cette précision accrue, on peut se retrouver dans des cas où la reproductibilité numérique est perdue, et ce totalement.

4 – Performance de la simulation

4.1 – Passage à l'échelle

Pour l'étude des performances de la simulation, nous avons travaillé sur une machine avec les spécifications matérielles suivantes : deux processeurs Intel Xeon E5-2650v2[33], avec 8 cœurs physiques et 16 cœurs logiques, cadencés à 2,60 GHz, ainsi qu'un Intel Xeon Phi 7220P[34] avec 61 cœurs physiques et 244 cœurs logiques, cadencés à 1,238 GHz.

Dans un premier temps, nous avons voulu évaluer les performances de la simulation avec ses différents paradigmes de parallélisation : plusieurs instances séquentielles en parallèle (appelées distribuée), ou bien une seule instance avec plusieurs threads en parallèle (appelée parallèle, en accord avec les définitions posées dans notre état de l'art). Nous avons comparé ces différents paradigmes sur les deux architectures matérielles à notre disposition.

[33] http://ark.intel.com/fr/products/75269/Intel-Xeon-Processor-E5-2650-v2-20M-Cache-2_60-GHz
[34] http://ark.intel.com/fr/products/75799/Intel-Xeon-Phi-Coprocessor-7120P-16GB-1_238-GHz-61-core

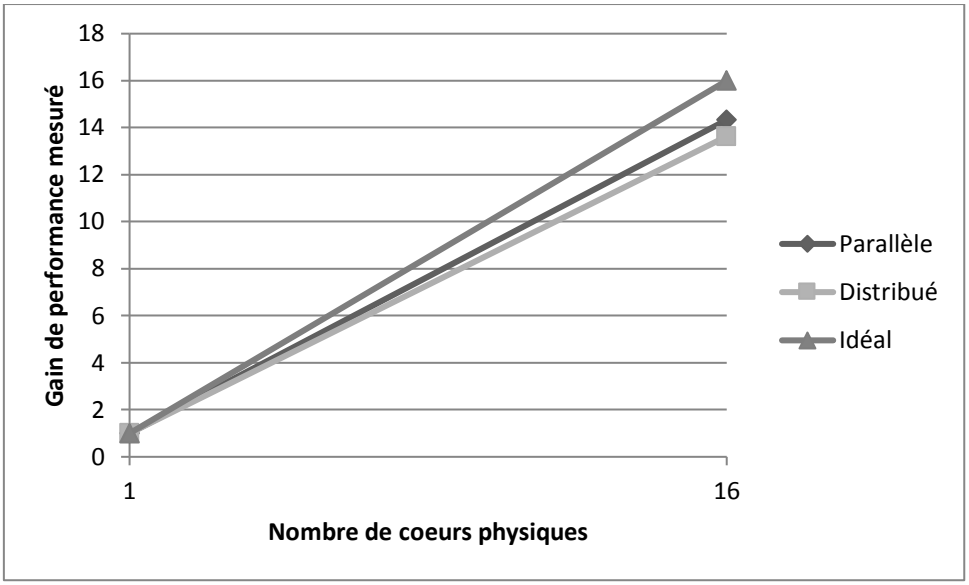

Figure 5-3 : Gain de performance lors de la distribution et la parallélisation sur les cœurs physiques de deux processeurs Intel Xeon

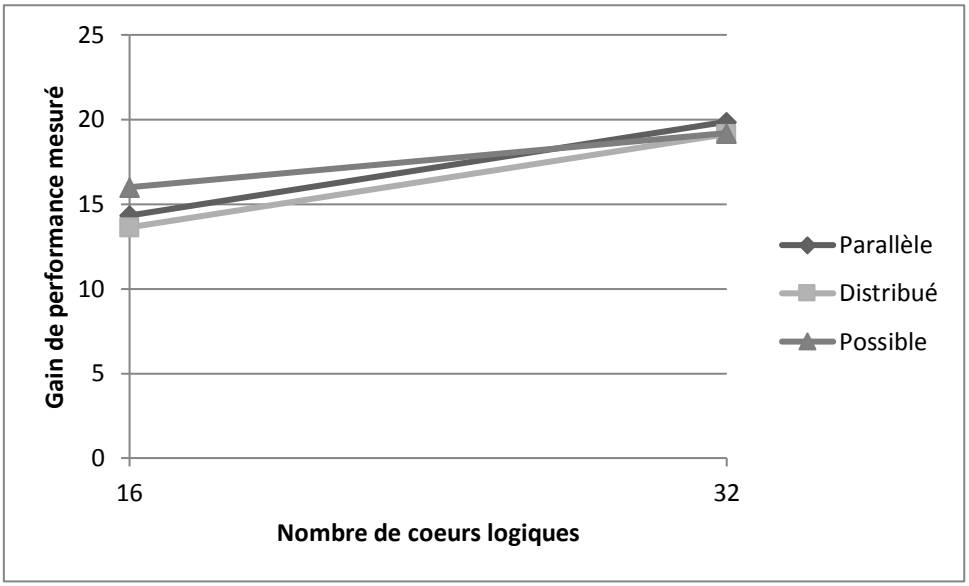

Figure 5-4 : Gain de performance lors de la distribution et la parallélisation sur les cœurs logiques de deux processeurs Intel Xeon

Sur la figure 1, nous pouvons constater que pour les modèles de parallélisme, tant que la simulation tourne sur les cœurs physiques des deux processeurs, le gain de performance est proche du gain de performance attendu idéal, attendu pour une simulation linéaire : 14,33X (contre 16X attendu). Puis, comme visible sur la figure 2, sur les 16 cœurs restant, la simulation est exécutée sur des cœurs

logiques dont la performance est moindre comparée à un pur cœur physique, c'est pourquoi on observe un fléchissement de la courbe. Cependant, sur cette figure 2, nous avons substitué la courbe « idéal » qui est clairement non accessible avec l'hyper-threading, par une courbe « possible » qui correspond à une performance d'hyper-threading de 20% par rapport à un cœur classique. On constate dès lors que la simulation devient super linéaire une fois le paradigme parallèle utilisé : la simulation exploite au mieux les ressources disponibles. Enfin, il apparaît que l'utilisation du parallélisme en lieu et place de la distribution entraine un gain de performance extrêmement limité, quasi imperceptible. On constate à peine 5% de gain sur les cœurs physiques, et jusqu'à 4% pour les cœurs logiques. Le seul avantage réel ici, puisque les performances sont proches, est la simplicité de lancement en cas de parallélisme, et l'absence de besoin de découper l'expérience et de fusionner les résultats à la fin des simulations. On pouvait néanmoins pressentir que le modèle distribué serait moins performant, en effet : la distribution revient donc à lancer 32 instances séquentielles de la simulation, les 32 instances lançant elles-mêmes deux threads : celui de calcul et celui d'écriture. Ce n'est clairement pas le modèle qui permettra de gagner en performance. Les différents threads d'écritures vont se gêner, et un thread de calcul n'aura plus le monopole de son cœur.

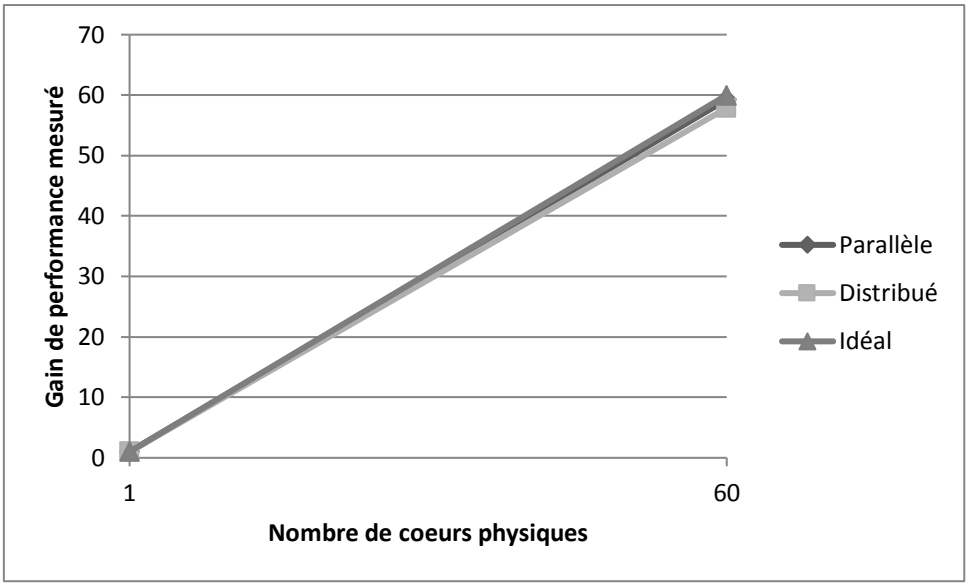

Figure 5-5 : Gain de performance lors de la distribution et la parallélisation sur les cœurs physiques d'un Xeon Phi

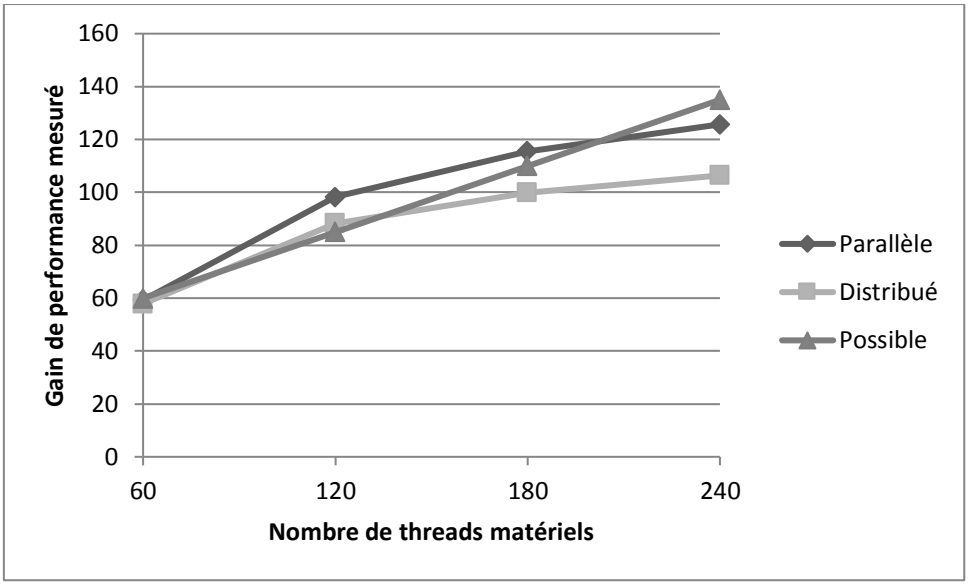

Figure 5-6 : Gain de performance lors de la distribution et la parallélisation sur les threads matériels d'un Xeon Phi

Sur la figure 3, nous pouvons constater que le passage à l'échelle sur Xeon Phi est encore meilleur que sur CPU. La simulation est vraiment très proche d'une simulation linéaire. Avec le modèle parallèle, on obtient un gain de performance de 59,26X, pour 60X attendu lorsque l'on travaille sur les cœurs physiques. Sur la figure 4, à l'image de ce qui pouvait être vu sur la figure 2 avec le CPU, on peut constater que les threads matériels fournissent moins de puissance à la simulation. Néanmoins, on peut constater que sous l'hypothèse qu'un thread matériel est 40% d'un cœur physique (c'est qui est toujours deux fois plus performant que l'hyper-threading des processeurs classiques), on peut constater que notre simulation arrive proche du gain de performance maximal possible (environ 126X, contre 135X possible). On peut également observer que jusqu'à 180 threads matériels, notre simulation, quel que soit le paradigme d'exécution choisi est super linéaire. Etant donné l'allure générale de la courbe, on peut d'ailleurs émettre l'hypothèse que plus le cœur physique est chargé, plus la performance de ses threads matériels va se réduire. C'est quelque chose de totalement nouveau par rapport à l'hyper-threading qui n'a que deux cœurs logiques et où on ne peut donc pas voir ce phénomène de charge. Contrairement au CPU, on peut constater que la différence entre distribué et parallèle est tangible, surtout sur les threads matériels. En effet, sur les cœurs physiques (figure 3), la différence entre les deux n'est que de 2,4% en faveur du CPU. Sur les threads matériels (figure 4), une fois tous exploités, elle est de 18% en faveur du CPU. C'était ici encore totalement prévisible, comme précédemment, la simulation parallèle optimise particulièrement ses entrées/sorties sur le disque, ce que ne fait pas la simulation distribuée où elles sont réalisées de

façon concurrente. Le Xeon Phi est particulièrement sensible aux entrées/sorties et n'est pas optimisé pour : les performances s'écrasent donc dès qu'elles deviennent trop nombreuses.

4.2 – Performances de l'hyper-threading

Nous avons par la suite voulu mesurer l'impact de l'hyper-threading sur notre simulation. Un tel impact a déjà été mesuré dans le passé (Leng et al., 2002), mais la conclusion des auteurs était que l'impact était fortement dépendent de la simulation et de son usage des cœurs de calcul. Afin de mettre en avant ceci, nous avons décidé de paralléliser la simulation petit-à-petit en utilisant un nombre croissant de cœurs : dans un premier temps, les cœurs physiques du premier CPU (cœurs de 1 à 8), puis les cœurs logiques de ce même CPU (cœurs de 9 à 16), puis les cœurs physiques du second CPU (cœurs 17 à 24), puis enfin les cœurs logiques du second CPU (cœurs 25 à 32). Nous avons mesuré le gain de performance à chacune de ces étapes. Les résultats sont affichés figure 5.

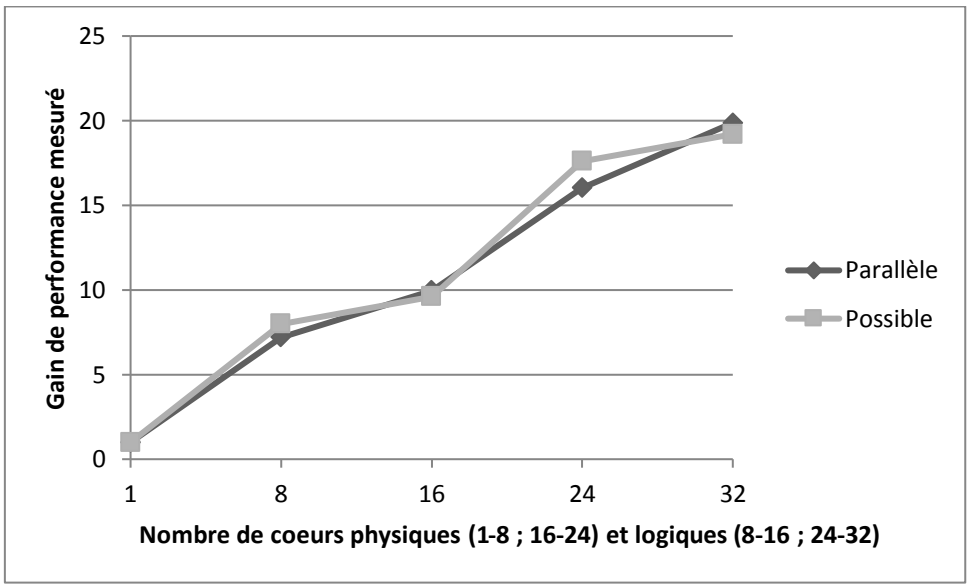

Figure 5-7 : Gain de performance lors de la parallélisation de la simulation d'abord sur le premier CPU puis sur le second

Sur la figure 5, on peut clairement voir deux pentes différentes pour nos gains de performance. La pente est bien plus forte lorsque l'on est sur les cœurs physiques des processeurs. La courbe possible, ici encore, prend en compte le facteur limitant du cœur logique (20% d'un cœur physique). Il apparaît clairement que lorsque l'on est sur les cœurs physiques, la pente est bien plus proche de

la pente idéale (gain de 1 par cœur ajouté). On peut ainsi vérifier que, comme énoncé précédemment, les cœurs logiques correspondant à l'hyper-threading sont bien moins performants que les cœurs physiques. Par ailleurs, on peut donc bien vérifier que sur les figures 1 et 3, on observe bien les cœurs physiques : l'ordonnance de Linux tente dans un premier temps de répartir sa charge sur les cœurs physiques avant d'allouer les cœurs logiques, une fois à court de cœurs physiques. Ainsi, les courbes observées sur les figures 2 et 4 ne sont pas un tassement des performances en raison d'un plus grand nombre d'instances lancées, mais tout simplement un tassement dû à l'usage des cœurs logiques, une fois tous les cœurs physiques exploités. Nous pouvons également conclure encore une fois que notre simulation arrive à exploiter plus de puissance que celle disponible *via* l'hyper-threading : elle est super linéaire. De plus, quand la simulation est parallélisée sur le second processeur, la différence entre cœur logique et cœur physique semble moins marquée, probablement en raison du partage des ressources entre les deux processeurs, et le fait que le second processeur doit obligatoirement passer par le premier pour accéder à certaines ressources (sauf la RAM).

4.3 – Comparaison des performances entre processeur Xeon et Xeon Phi

Finalement, nous nous sommes intéressés aux performances comparatives de la simulation une fois parallélisée sur CPU et sur Phi. Pour ce test, nous avons procédé de la façon suivante : nous avons d'abord exploité tous les cœurs disponibles sur le Xeon Phi, et sur un des deux processeurs. Puis, nous avons exploité tous les cœurs disponibles sur le Xeon Phi et sur les deux processeurs. Dans les deux cas, nous avons comparés la même simulation (même nombre d'évènements, même initialisation). Nous avons simulés un milliard d'évènements afin d'avoir un temps d'exécution conséquent. Les résultats sont donnés dans le tableau 4.

Tableau 5-4: Performance d'une simulation à 1 milliard d'évènements parallélisée sur 1 Phi, 1 CPU, 2 CPUs

	Intel Xeon Phi 7120P	Intel Xeon E5-2650v2	2x Intel Xeon E5-2650v2
Temps	48 h 49 m	36 h 32 min	18 h 17 min
Gain de performance	1X	1.34X	2.67X

A la vue des résultats du tableau 4, la première observation que l'on peut faire relève de l'évidence : sur cette application où l'aspect prédominant est le calcul, le Xeon Phi est plus lent qu'un CPU, et donc, que de deux CPUs. Les performances du Xeon Phi sont particulièrement mauvaises : alors que le seul Xeon Phi a besoin d'un peu plus de deux jours pour mener à bien la tâche, les deux processeurs peuvent la régler en moins d'une journée.

Il est intéressant de remettre ces résultats en perspective avec ce qui a été exposé dans le chapitre 4 sur les simulations *memory-bound* et plus en détails dans (Schweitzer et al., 2015). Nous avons notamment montré qu'il était possible sur une autre catégorie d'application, d'avoir un gain de performance de l'ordre de 2,6X en faveur du Xeon Phi par rapport à notre modèle de CPU Ivy Bridge v2 à 2,6 Ghz, sous la contrainte spécifique d'avoir une application *memory-bound*. Nous avons pu constater ce phénomène en distribuant à la fois l'application sur le Xeon Phi et sur nos deux CPUS Ivy Bridge v2. Sur les deux CPUs, le gain de performance était limité à environ 2X, alors qu'on aurait pu s'attendre à un gain de l'ordre de 20X, tandis que sur le Xeon Phi, qui une bande passante-mémoire bien plus importante (environ 7 fois plus importante), le goulet d'étranglement lié à la consommation mémoire était effacé, et on pouvait atteindre des gains de performance de l'ordre de 156X. En revanche, ici, nous pouvons constater que quand rien n'empêche de consommer l'intégralité des ressources du processeur, le Xeon Phi ne fournit pas assez de puissance. Les deux conclusions ne sont pas opposées mais complémentaires : dans le cas d'une application *memory-bound*, le Xeon Phi offrira de biens meilleures performances que le CPU. En revanche, pour une application *compute-bound*, le CPU offre de meilleures performances. Dans le cadre de cette thèse, essentiellement liée au calcul, nous n'avons pas étudié les applications intensives au niveau des entrées sorties.

Ce type de comparaisons peut être utilisé pour déterminer de quel type de goulet d'étranglement souffre une application. Par exemple, dans (Liu, Xu, & Carothers, 2014) est présenté une comparaison entre les performances d'une simulation de physique nucléaire entre un processeur Intel X5650[35] et un Xeon Phi 5110P. Les auteurs obtiennent un gain de performance de l'ordre de 3,3X en faveur du 5110P. Leurs résultats montrent qu'ils ont une application *memory-bound*, ce qui est d'ailleurs, le cas le plus fréquent. Cependant, ce n'est pas toujours le cas : notre simulation est l'exemple typique d'une simulation *compute-bound*, d'où une parallélisation plus efficace sur des CPU que sur des Xeon Phis.

[35] http://ark.intel.com/fr/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI

5 – Conclusions

Dans ce chapitre, nous avons décrit et analysé une simulation « preuve de concept » aux propositions formulées dans le chapitre 3. Nous avons réalisé une simulation capable de passer à l'échelle, qui fournit des résultats statistiquement corrects et numériquement reproductibles.

Nous avons en effet d'abord évalué la reproductibilité numérique de la simulation en utilisant une bonne technique de parallélisation des flux stochastiques. D'abord en changeant les paradigmes d'exécution : parallèle, distribué, séquentiel (tels que définit dans le chapitre de l'état de l'art). Une fois assuré que cette reproductibilité numérique était acquise, ainsi que la reproductibilité « *run-to-run* », nous nous sommes intéressés à la reproductibilité numérique entre deux architectures matérielles différentes, le processeur classique et le Xeon Phi d'Intel. D'abord sans prêter aucune attention à la mise en œuvre des bonnes pratiques de compilation, ce qui a conduit à des différences catastrophiques puis entre les deux architectures, et enfin en suivant les pratiques rigoureuses présentées au chapitre 3. En appliquant ces pratiques, nous obtenons une reproductibilité numérique plus que satisfaisante : elle est bit-à-bit entre les deux architectures pour une grande majorité des valeurs en double précision, et pour toutes les valeurs en simple précision.

Ensuite, nous avons évalué les performances de notre simulation. Notre simulation est purement calculatoire et est donc *compute-bound*. Nous avons ainsi pu constater que notre simulation présente un gain de performance quasi linéaire sur les cœurs physiques d'un processeur et super linéaire sur les cœurs logiques du processeur. On observe la même tendance sur les cœurs physiques puis sur les threads matériels du Xeon Phi. Nous en avons profité pour évaluer le comportement de l'ordonnanceur, pour vérifier qu'il envoie d'abord les instances sur les cœurs physiques, puis une fois tous utilisés, sur les cœurs logiques. Nous avons néanmoins pu confirmer que lorsqu'une application est *compute-bound*, le Xeon Phi présente de moins bonnes performances : un processeur de dernière génération (Ivy Bridge v2), sans avoir une fréquence crête trop importante (2,6 Ghz), présente un gain de performance de 1,34X face au Xeon Phi 7120P.

Conclusions

Cette thèse s'est inscrite dans le contexte du calcul à haute performance, appliqué à la tomographie de larges édifices à l'aide de muons. Tandis qu'à l'aide d'un détecteur, on est capable de mesurer la transmittance de muons à travers une structure, telle qu'un volcan, il est nécessaire de pouvoir reproduire ce phénomène physique informatiquement, dans des simulations afin de valider notre bonne compréhension de ce que l'on mesure. Parce que l'on travaille sur des processus purement stochastiques, ces simulations sont implémentées sous la forme de simulations de Monte Carlo. Nous avons donc, dans le cadre de cette thèse travaillé sur la mise au point, l'optimisation et la parallélisation de telles simulations de Monte Carlo. Ces simulations de Monte Carlo ont été à la fois développées pour processeurs classiques et pour accélérateurs matériels de type Xeon Phi. Dès lors, la question de la reproductibilité s'est posée. A travers ces simulations de Monte Carlo, nous avons pu émettre plusieurs propositions que nous avons mises en œuvre à travers la thèse.

1 – Résultats

Dans un premier temps, nous avons repris une simulation de Monte Carlo, `tomusim`. A l'origine, écrite en Java et en Python, elle présentait des performances plus que mauvaises : elle était plus lente que le phénomène physique simulé. Nous avons donc fortement optimisé cette simulation, notamment en abandonnant le Python, pour du C++, en optimisant les communications entre les objets Java et le C++ grâce à la Java Native Interface, qui se sont révélées grâce au profilage de la simulation, le goulet d'étranglement principal de la simulation. Grâce à cette optimisation, et d'autres optimisations d'ordre algorithmique, directement sur le modèle physique, nous avons ainsi pu obtenir un gain de performance de plus de 400X en parallélisant la simulation sur un nœud de calcul avec 32 cœurs physiques. Ce gain de performance, qui a d'ailleurs permis d'avoir une simulation plus rapide que le phénomène physique, a autorisé l'implémentation de nouvelles fonctionnalités visant à améliorer la précision de la simulation.

Nous nous sommes ensuite penchés sur l'écriture d'un profileur programmé en C++, à l'aide du paradigme de programmation orientée aspect. Celui-ci avait été écrit à l'origine par palier les problèmes rencontrés lors de l'utilisation de `valgrind` sur la simulation `tomusim`. Etant donné que les premiers résultats obtenus par ce profileur étaient particulièrement satisfaisants, nous avons poursuivi l'implémentation en rajoutant des fonctionnalités permettant le profilage d'applications parallèles, s'appuyant sur la bibliothèque `pthread`, mais également le profilage d'applications

s'exécutant sur des architectures parallèles (telles que *manycore* ou *multicore*). Malgré des performances en-deçà de profileurs tels que `valgrind`, nous avons pu constater que notre profileur supporte parfaitement le parallélisme d'applications écrites avec la bibliothèque `pthread`, et mieux encore, établi correctement les coûts et les arbres d'appels par thread. Il supporte également, contrairement à `valgrind`, le parallélisme de la simulation au niveau du système d'exploitation : plutôt que d'écraser l'application sur un unique cœur de calcul (ce qui peut fausser les résultats), il travaille bien sur tous les cœurs exploités par l'application. Le code source de ce profileur a par ailleurs été libéré[36].

Enfin, nous avons développé une simulation parallèle de Monte Carlo pour la propagation de muons à travers un bloc de matière, nommée `parsitomu`. Cette simulation a été développée à partir de rien, et sans l'utilisation d'un quelconque cadriciel. Cela était nécessaire afin de pouvoir répondre au cahier des charges qui avait été fixé : simulation parallèle, qui passe à l'échelle, qui fournit des résultats statistiquement viables et qui soit numériquement reproductible. Nous avons développé cette simulation en C++ avec la bibliothèque `pthread`. L'axe central de cette simulation est ses flux stochastiques, qui servent de référentiel pour les particules. Afin que ces flux restent reproductibles dans toutes les conditions, nous nous sommes appuyés sur un générateur robuste et reproductible, fournit avec son propre algorithme de « *jump-ahead* » facilement utilisable dans une application : MRG2k3a, qui est directement embarqué dans le code de la simulation pour ne pas créer de dépendance supplémentaire. Un objectif de cette simulation étant qu'elle soit suffisamment légère pour pouvoir être facilement déplacée d'une architecture à une autre : *multicore* et *manycore*. Nous avons d'abord fait une analyse automatisée de la simulation : celle-ci ne présente pas de problème de concurrence dans son implémentation parallèle. Ce résultat est important, puisqu'il évite une des sources de non reproductibilité. Puis, nous avons évalué la reproductibilité de cette simulation. Cette simulation est reproductible bit-à-bit d'une exécution à l'autre lorsqu'elle est initialisée de la même façon. Mieux, quel que soit le paradigme d'exécution choisi, si les conditions initiales sont les mêmes, elle donne les mêmes résultats (éventuellement dans le désordre). Nous nous sommes alors intéressés à la reproductibilité numérique entre les architectures Xeon Phi et Xeon classique. Comme Intel l'a annoncé, nous n'avons pas toujours observé une reproductibilité bit-à-bit. Nous pouvions constater quelques divergences sur certaines valeurs. Cependant, ces divergences semblent être majorées, et l'erreur est toujours inférieure à la précision d'un nombre à virgule flottante en simple-précision. C'est-à-dire que si l'on considère les résultats en simple-précision, nous avons une reproductibilité numérique bit-à-bit entre les deux architectures. C'est un résultat majeur et très important. Quand on considère le modèle sur lequel on travaille, étant donné le bruit

[36] https://github.com/HeisSpiter/acprof

de fond inhérent à l'observation physique (détecteur, muons « parasites », etc.), la reproductibilité numérique bit-à-bit en simple-précision est excellente. C'est donc un premier objectif de la simulation atteint, haut la main. Nous avons également profité de cette excellente reproductibilité numérique pour explorer l'impact des options de compilation pour la simulation. Nous avons évalué la divergence des résultats lorsque l'on utilise des options de compilations classiques : on se retrouve avec des différences importantes, voire totalement catastrophique au point où deux résultats peuvent totalement diverger dans leur interprétation physique même entre les deux architectures. Nous avons constaté à quel point il est essentiel de prêter une attention particulière aux options de compilation favorisant la reproductibilité numérique. Ces options ont un coût en temps de calcul, mais à quoi bon calculer plus vite pour donner des résultats faux. Concernant les performances de la simulation, nous avons pu mesurer que sur les cœurs physiques d'un processeur classique, la simulation est quasiment linéaire. Mais mieux encore, une fois les cœurs logiques utilisés (avec leur limite de puissance prise en compte), la simulation est super-linéaire. Sur un Xeon Phi, la simulation se comporte encore mieux : elle est linéaire sur les cœurs physiques, et super linéaire sur les threads matériels (cœurs logiques). On constate d'ailleurs, que conformément à notre proposition, elle est plus performante sur processeur classique que sur Xeon Phi, étant donné que la simulation n'est pas « *memory-bound* ». Quoi qu'il en soit, nous avons atteint nos objectifs : cette simulation est numériquement reproductible, et elle fournit d'excellentes performances tout en étant portable et en passant à l'échelle (nous simulons un milliard de particules en un temps tout à fait raisonnable). Elle résume à elle-seule l'intégralité du savoir-faire acquis lors de cette thèse.

En plus de ces implémentations et de ces travaux importants, nous avons pu glaner d'autres résultats. En effet, nous avons pu établir que parce que le Xeon Phi a une bande passante mémoire bien plus importante qu'un processeur classique, il est la plate-forme idéale pour l'exécution des applications qui sont « *memory-bound* ». C'est ainsi que nous avons pu avoir un gain de performance de 2,56X pour une simulation développée avec `Geant4`, `tmvg4sim`, sur le Xeon Phi, par rapport à un processeur classique, la simulation ne pouvant pas accéder assez rapidement à ses données sur processeur classique. De plus, nous avons aussi pu montrer que l'utilisation de KSM, même si il avait été pensé à l'origine pour le monde de la virtualisation, a toute sa place dans le monde du calcul à haute performance, notamment sur les nœuds de calcul où la mémoire est en quantité limitée, telle que les Xeon Phi. Nous avons ainsi pu établir que pour notre simulation `tmvg4sim`, l'utilisation de KSM sur un Xeon Phi 5110P, permettait de libérer environ 8% de la RAM et par conséquent, de lancer 20 instances de la simulation supplémentaires, soit environ 16% d'instances supplémentaires. Nous avons pu également constater que le coût en temps de calcul provoqué par l'utilisation de KSM était suffisamment minime pour justifier son usage dans le calcul à haute performance. Enfin, nous

avons pu mesurer l'impact des instructions vectorielles sur les performances d'une application. En effet, nous avons pu constater que les performances de ces instructions ne sont pas égales selon les gammes de processeurs que l'on considère. Si sur les processeurs grand public, de type Intel Core, les instructions vectorielles permettent de gagner en performance, sur les processeurs prévus pour les serveurs, de type Intel Xeon, les instructions vectorielles peuvent impact négativement les performances, notamment dans le cas de l'utilisation des vectorisations automatiques des compilateurs.

2 – Perspectives

A l'issue de ces travaux, de ces résultats, mais également des avancées dans le domaine du calcul à haute performances, de nouvelles perspectives peuvent être ouvertes sur ce qui a été réalisé lors de cette thèse. La première des perspectives que l'on peut ouvrir est celle du « *Green Computing* ». Aujourd'hui, pour avoir des performances, on consomme plus de watts, et on se félicite du gain de performances. Cependant, la question se pose : comment atteindre l'*exascale*, i.e., l'instant où l'on aura un super-ordinateur avec une puissance de calcul de l'ordre de l'exa-FLOPS si l'on consomme autant ? Une réponse est de consommer moins, et c'est ainsi que le « *Green Computing* » commence à arriver dans le monde du calcul à haute performance : tenter de conserver nos performances actuelles en consommant moins, et ce, grâce aux architectures matérielles de type ARM. Lors de notre thèse, nous nous sommes attachés à utiliser des technologies qui consomment et qui sont toujours à base d'x86 d'Intel. Mais cela pose alors la question suivante : quelles auraient été les performances de nos simulations si nous avions utilisés un cluster de machines ARM ? Mais également, quelle reproductibilité numérique aurions-nous si nous exécutions nos simulations sur un processeur x86 et sur un processeur ARM ? Une fois ces questions abordées, nous pouvons pousser le questionnement encore plus en profondeur pour le « *Green Computing* » : combien faudrait-il de nœuds ARM pour avoir la puissance de calcul d'un Xeon Phi ou d'un bon processeur x86 comme ceux que nous avons exploités durant cette thèse ? Et une fois tous ces nœuds branchés et en train de calculer, quelle serait réellement la puissance consommée ? In fine, quel serait le véritable gain en énergie, pour quel coût en développement et en performance ? Ces questions à elles-seules, par leur nombre et par leur complexité justifieraient d'une nouvelle thèse sur le sujet. A noter que la simulation `parsitomu` a également été développée pour aider à répondre à ces questions : elle est suffisamment légère et parallèle pour pouvoir être exécutée sur un cluster de nœuds de calcul ARM. Et son format de fichier suffisamment simple pour permettre une évaluation de la reproductibilité numérique entre ces différentes architectures.

Ce questionnement est très important, puisque pour certains, il conditionne l'avenir du calcul haute performance : nous ne pourrons pas continuer à grossir si nous ne consommons pas moins. Encore faut-il le faire proprement, en conservant la reproductibilité de nos résultats. Mais ce n'est pas le seul questionnement qu'on peut avoir une fois cette thèse lue. On peut se poser la question de la place du GP-GPU. Intel a-t-il véritablement fourni un concurrent pour le GP-GPU avec ses Xeon Phi tels qu'on les connaît au moment de la rédaction ? Comme nous avons pu le démontrer au fil de notre thèse, les promesses faites par Intel sur le gain de performance que l'on peut tirer « facilement » avec le Xeon Phi ne sont pas si évidentes que cela. Si nous avons pu effectivement obtenir de bons gains de performance pour les applications « *memory-bound* », ça n'est pas du tout le cas pour les applications « *compute-bound* ». Est-ce que le GP-GPU aurait fait mieux que le Xeon Phi pour ces applications ? Qu'en aurait-il été de la reproductibilité de nos résultats ? On pourrait s'imaginer reprendre la publication sur `parsitomu`, sa reproductibilité, ses performances pour l'étendre et prendre en compte le GP-GPU et l'architecture ARM pour faire une comparaison quadripartite tant d'un point de vue performance que d'un point de vue reproductibilité. Le tout en gardant à l'esprit que la nouvelle forme du Xeon Phi, SoC (*System on Chip*) pourrait révolutionner le Xeon Phi et ce qu'on en tire à l'heure actuelle. En effet, cette nouvelle implémentation du Xeon Phi (nom de code Knights Landing) sera un « *reboot* » du Xeon Phi : disponible sous forme d'une socket, réutilisant l'architecture x86_64 (abandon de l'architecture propre `k1om`) et disposant de jeux d'instructions communs à la prochaine génération de CPU (Skylane) mais également d'instructions propres pour le calcul à haute performance.

Si l'on se recentre sur l'aspect logiciel de la thèse, on peut s'interroger sur les perspectives même à donner à `parsitomu`. Si `parsitomu` représente le point d'orgue de cette thèse en répondant à un cahier des charges des plus rigoureux autant algorithmiquement que théoriquement, ne serait-il pas possible d'en extraire le cœur, le moteur de simulation de `parsitomu`, afin de le transformer lui-même en cadriciel de simulation, ouvert à n'importe quel scientifique, désireux de faire de la simulation parallèle de Monte Carlo reproductible et qui passe à l'échelle ? Le tout en lui rendant le travail suffisamment simple pour qu'il n'ait pas besoin de se concentrer sur les aspects purement techniques qui font la force de `parsitomu`. Sur ces aspects techniques, on peut se poser une question : à l'heure actuelle, on travaille de plus en plus sur les algorithmes parallèles dit « *lock-free* ». C'est-à-dire qu'ils sont capable de gérer plusieurs threads en parallèle sur leurs données, sans besoin de synchronisation. Serait-il possible d'envisager une réduction des mécanismes de verrouillage dans `parsitomu` pour augmenter encore son efficacité parallèle ? Ceux-ci ne servant déjà qu'à synchroniser les mises en attente des écritures et les initialisations des flux stochastiques.

Au sujet des flux stochastiques, nous avons donné des méthodes très rigoureuses, autant dans l'état de l'art que dans les propositions (et ensuite dans les mises en application) sur leur gestion. Mais dans notre cadre expérimental propre, nous n'avons jamais évalué l'impact d'un mauvais générateur (type LCG) ou d'une mauvaise gestion des flux sur la viabilité statistique des résultats des simulations. Quel serait l'impact de ces « mauvaises » gestions ? Serait-il pire que le bruit ambiant observé lors des mesures physiques ? Ou passerait-il inaperçu une fois replacé dans le contexte physique considéré ? Il serait intéressant d'évaluer l'impact, les distributions des fonctions de densités des résultats dans ces cas, cela permettrait de répondre à ces questions.

Enfin, une dernière perspective que nous avons très rapidement évoquée dans la thèse : nous nous sommes intéressés aux applications *compute-bound* (`parsitomu`), *memory-bound* (`tmvg4sim`), mais très peu aux applications *I/O-bound*. On pourrait éventuellement considérer `tomusim` comme une application *I/O-bound*. Nous avons effectivement rencontré des problèmes liés aux entrées/sorties lors de l'optimisation de cette simulation, mais surtout centrés sur l'initialisation et la conclusion de la simulation, pas lors de la partie purement calculatoire. Ainsi, nous n'avons pas réellement de données de performances sur ce troisième type d'applications qui passent la majeure partie de leur temps à lire et à écrire des données sur le disque. Encore moins sur Xeon Phi, étant donné que `tomusim` n'a jamais été portée sur Xeon Phi, notamment en raison de la dépendance à Java. De façon générale, une attention aurait pu être portée sur l'impact des supports de stockage sur les performances des applications notamment avec l'émergence de nouvelles stratégies de stockage et de périphériques de stockages. Cependant, faute d'application réellement *I/O-bound*, cela se serait révélé plus complexe.

Références bibliographiques

Aas, J. (2005). *Understanding the Linux 2.6.8.1 CPU Scheduler*. Retrieved from http://retep.googlecode.com/svn-history/r381/trunk/IR/Biblio_Android/linuxKernelUnderstandingQueudet.pdf

Abrahams, D., & Gurtovoy, A. (2004). *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Pearson Education.

Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., & Tallent, N. R. (2010). HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, *22*(6), 685–701.

Agostinelli, S., Allison, J., Amako, K., Apostolakis, J., Araujo, H., Arce, P., Asai, M., Axen, D., Banerjee, S., Barrand, G., Behner, F., Bellagamba, L., Boudreau, J., Broglia, L., Brunengo, A., Burkhardt, H., Chauvie, S., Chuma, J., Chytracek, R., Cooperman, G., Cosmo, G., Degtyarenko, P., Dell'Acqua, A., Depaola, G., Dietrich, D., Enami, R., Feliciello, A., Ferguson, C., Fesefeldt, H., Folger, G., Foppiano, F., Forti, A., Garelli, S., Giani, S., Giannitrapani, R., Gibin, D., Cadenas, J. J. G., González, I., Abril, G. G., Greeniaus, G., Greiner, W., Grichine, V., Grossheim, A., Guatelli, S., Gumplinger, P., Hamatsu, R., Hashimoto, K., Hasui, H., Heikkinen, A., Howard, A., Ivanchenko, V., Johnson, A., Jones, F. W., Kallenbach, J., Kanaya, N., Kawabata, M., Kawabata, Y., Kawaguti, M., Kelner, S., Kent, P., Kimura, A., Kodama, T., Kokoulin, R., Kossov, M., Kurashige, H., Lamanna, E., Lampén, T., Lara, V., Lefebure, V., Lei, F., Liendl, M., Lockman, W., Longo, F., Magni, S., Maire, M., Medernach, E., Minamimoto, K., Freitas, P. M. de, Morita, Y., Murakami, K., Nagamatu, M., Nartallo, R., Nieminen, P., Nishimura, T., Ohtsubo, K., Okamura, M., O'Neale, S., Oohata, Y., Paech, K., Perl, J., Pfeiffer, A., Pia, M. G., Ranjard, F., Rybin, A., Sadilov, S., Salvo, E. Di, Santin, G., Sasaki, T., Savvas, N., Sawada, Y., Scherer, S., Sei, S., Sirotenko, V., Smith, D., Starkov, N., Stoecker, H., Sulkimo, J., Takahata, M., Tanaka, S., Tcherniaev, E., Tehrani, E. S., Tropeano, M., Truscott, P., Uno, H., Urban, L., Urban, P., Verderi, M., Walkden, A., Wander, W., Weber, H., Wellisch, J. P., Wenaus, T., Williams, D. C., Wright, D., Yamada, T., Yoshida, H., & Zschiesche, D. (2003). GEANT4 - a simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, *506*(3), 250–303.

Akdemir, K., Dixon, M., Feghali, W., Fay, P., Gopal, V., Guilford, J., Ozturk, E., Wolrich, G., & Zohar, R. (2010). Breakthrough AES Performance with Intel AES New Instructions. *Intel White Paper*.

Allen, F. E. (1974). Interprocedural Data Flow Analysis. In *Proc. IFIP Congress 74* (pp. 398–402). Amsterdam: North-Holland Publishing Co.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)* (pp. 483–485). New York, New York, USA: ACM Press. http://doi.org/10.1145/1465482.1465560

Ansaloni, D., Binder, W., Villazón, A., & Moret, P. (2010). Parallel dynamic analysis on multicores with aspect-oriented programming. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development* (pp. 1–12). ACM.

Apel, S., Leich, T., Rosenmüller, M., & Saake, G. (2005). FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. *Generative Programming and Component Engineering*, 125–140.

Arcangeli, A., Eidus, I., & Wright, C. (2009). Increasing memory density by using KSM. In *Proceedings of the linux symposium* (pp. 19–28).

AspectC: AOP for C. (2004). Retrieved from http://www.aspectc.net

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., & Weeratunga, S. K. (1991). The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, *5*(3), 63–73.

Balakrishnan, S., Rajwar, R., Upton, M., & Lai, K. (2005). The impact of performance asymmetry in emerging multicore architectures. *ACM SIGARCH Computer Architecture News*, *33*(2), 506–517. Retrieved from http://dl.acm.org/citation.cfm?id=1070012

Barthou, D., Rubial, A. C., Jalby, W., Koliai, S., & Valensi, C. (2010). Performance tuning of x86 openmp codes with maqao. In *Tools for High Performance Computing 2009* (pp. 95–113). Springer.

Ben-Kiki, O., Evans, C., & Ingerson, B. (2009). *YAML Ain't Markup Language (YAML™) Version 1.2*. Retrieved from http://www.yaml.org/spec/1.2/spec.html

Berg, S. G. (2002). Cache prefetching. In *Techique Report UW-CSE 02-02-04*.

Beringer, J., Arguin, J.-F., Barnett, R. M., Copic, K., Dahl, O., Groom, D. E., Lin, C.-J., Lys, J., Murayama, H., Wohl, C. G., Yao, W.-M., Zyla, P. A., Amsler, C., Antonelli, M., Asner, D. M., Baer, H., Band, H. R., Basaglia, T., Bauer, C. W., Beatty, J. J., Belousov, V. I., Bergren, E., Bernardi, G., Bertl, W., Bethke, S., Bichsel, H., Biebel, O., Blucher, E., Blusk, S., Brooijmans, G., Buchmueller, O., Cahn, R. N., Carena, M., Ceccucci, A., Chakraborty, D., Chen, M.-C., Chivukula, R. S., Cowan, G., D'Ambrosio, G., Damour, T., de Florian, D., de Gouvêa, A., DeGrand, T., de Jong, P., Dissertori, G., Dobrescu, B., Doser, M., Drees, M., Edwards, D. A., Eidelman, S., Erler, J., Ezhela, V. V., Fetscher, W., Fields, B. D., Foster, B., Gaisser, T. K., Garren, L., Gerber, H.-J., Gerbier, G., Gherghetta, T., Golwala, S., Goodman, M., Grab, C., Gritsan, A. V., Grivaz, J.-F., Grünewald, M., Gurtu, A., Gutsche, T., Haber, H. E., Hagiwara, K., Hagmann, C., Hanhart, C., Hashimoto, S., Hayes, K. G., Heffner, M., Heltsley, B., Hernández-Rey, J. J., Hikasa, K., Höcker, A., Holder, J., Holtkamp, A., Huston, J., Jackson, J. D., Johnson, K. F., Junk, T., Karlen, D., Kirkby, D., Klein, S. R., Klempt, E., Kowalewski, R. V., Krauss, F., Kreps, M., Krusche, B., Kuyanov, Y. V., Kwon, Y., Lahav, O., Laiho, J., Langacker, P., Liddle, A., Ligeti, Z., Liss, T. M., Littenberg, L., Lugovsky, K. S., Lugovsky, S. B., Mannel, T., Manohar, A. V., Marciano, W. J., Martin, A. D., Masoni, A., Matthews, J., Milstead, D., Miquel, R., Mönig, K., Moortgat, F., Nakamura, K., Narain, M., Nason, P., Navas, S., Neubert, M., Nevski, P., Nir, Y., Olive, K. A., Pape, L., Parsons, J., Patrignani, C., Peacock, J. A., Petcov, S. T., Piepke, A., Pomarol, A., Punzi, G., Quadt, A., Raby, S., Raffelt, G., Ratcliff, B. N., Richardson, P., Roesler, S., Rolli, S., Romaniouk, A., Rosenberg, L. J., Rosner, J. L., Sachrajda, C. T., Sakai, Y., Salam, G. P., Sarkar, S., Sauli, F., Schneider, O., Scholberg, K., Scott, D., Seligman, W. G., Shaevitz, M. H., Sharpe, S. R., Silari, M., Sjöstrand, T., Skands, P., Smith, J. G., Smoot, G. F., Spanier, S., Spieler, H., Stahl, A., Stanev, T., Stone, S. L., Sumiyoshi, T., Syphers, M. J., Takahashi, F., Tanabashi, M., Terning, J., Titov, M., Tkachenko, N. P., Törnqvist, N. A., Tovey, D., Valencia, G., van Bibber, K., Venanzoni, G., Vincter, M. G., Vogel, P., Vogt, A.,

Walkowiak, W., Walter, C. W., Ward, D. R., Watari, T., Weiglein, G., Weinberg, E. J., Wiencke, L. R., Wolfenstein, L., Womersley, J., Woody, C. L., Workman, R. L., Yamamoto, A., Zeller, G. P., Zenin, O. V., Zhang, J., Zhu, R.-Y., Harper, G., Lugovsky, V. S., & Schaffner, P. (2012). Review of Particle Physics. *Physical Review D*, *86*(1), 010001. http://doi.org/10.1103/PhysRevD.86.010001

Bernaschi, M., Bisson, M., & Salvadore, F. (2014). Multi-Kepler GPU vs. multi-Intel MIC for spin systems simulations. *Computer Physics Communications*, *185*(10), 2495–2503. http://doi.org/10.1016/j.cpc.2014.05.026

Bethe, H., & Heitler, W. (1934). On the stopping of fast particles and on the creation of positive electrons. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, *146*(856), 83–112.

Bhandarkar, D., & Ding, J. (1997). Performance characterization of the Pentium Pro processor. In *High-Performance Computer Architecture, 1997., Third International Symposium on* (pp. 288–297). IEEE.

Binder, W., Ansaloni, D., Villazón, A., & Moret, P. (2011). Flexible and efficient profiling with aspect-oriented programming. *Concurrency Computation Practice and Experience*, *23*(15), 1749–1773. http://doi.org/10.1002/cpe.1760

Bogdanov, A., Burkhardt, H., Ivanchenko, V., Kelner, S., Kokoulin, R., Maire, M., Rybin, A., & Urban, L. (2006). Geant4 simulation of production and interaction of muons. *Nuclear Science, IEEE Transactions on*, *53*(2), 513–519.

Brandfass, B., Alrutz, T., & Gerhold, T. (2013). Rank reordering for MPI communication optimization. *Computers and Fluids*, *80*, 372–380. http://doi.org/10.1016/j.compfluid.2012.01.019

Brun, R., & Rademakers, F. (1997). ROOT—an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics …*, *389*, 81–86. Retrieved from http://www.sciencedirect.com/science/article/pii/S016890029700048X

Brunner, T. A. (2003). Cell Based Random Number Generator For Reproducible Monte Carlo Simulations. *American Nuclear Society Topical Meeting in Mathematics & Computations*.

Büttner, D., & Weidendorfer, J. (2013). Analysis and Optimization of the Memory Access Behavior of Applications. In *Ecole "optimisation"* (p. 67). Saclay. Retrieved from http://calcul.math.cnrs.fr/IMG/pdf/DavidBuettner-AnalysisandOptimizationoftheMemoryAccess.pdf

Calafiura, P., Eranian, S., Levinthal, D., Kama, S., & Vitillo, R. A. (2012). GOoDA: The Generic Optimization Data Analyzer. *Journal of Physics: Conference Series*, *396*(5), 052072. http://doi.org/10.1088/1742-6596/396/5/052072

Callgrind: a call-graph generating cache and branch prediction profiler. (2013). Retrieved September 26, 2013, from http://valgrind.org/docs/manual/cl-manual.html

Chandy, K. M., & Misra, J. (1979). Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, *5*, 440–452.

Chauhan, N., Kabra, G., Kittelmann, T., Langenberg, R., Mandrysch, R., Salzburger, A., Seuster, R., Ritsch, E., Stewart, G., van Eldik, N., & Vitillo, R. (2013). *ATLAS Offline Software Performance Monitoring and Optimization*. Retrieved from http://cds.cern.ch/record/1622268/files/ATL-SOFT-PROC-2013-040.pdf?version=1

Chekanov, S. (2008). HEP data analysis using jHepWork and Java. *Proceedings of the HERA-LHC Workshops*. Retrieved from http://arxiv.org/abs/0809.0840v2

Chirkin, D., & Rhode, W. (2001). Muon Monte Carlo: a new high-precision tool for muon propagation through matter. In *Proceedings of ICRC* (pp. 1017–1020). Retrieved from http://www.hepg.sdu.edu.cn/Chinese/literatures/27icrc/ICRC2001/papers/ici7356_p.pdf

Chirkin, D., & Rhode, W. (2004). Propagating leptons through matter with Muon Monte Carlo (MMC). *arXiv Preprint Hep-ph/0407075*. Retrieved from http://arxiv.org/abs/hep-ph/0407075

Chrysos, G. (2012). Intel xeon phi coprocessor (codename knights corner). In *Proceedings of the 24th Hot Chips Symposium, HC*.

Claerbout, J. (1990). Active documents and reproducible results. *Stanford Exploration Project Report*, *67*, 139–144.

Claerbout, J., & Karrenbach, M. (1992). Electronic documents give reproducible research a new meaning. In *Proceedings of the 62nd Annual International Meeting of the Society of Exploration Geophysics*.

Coady, Y., & Kiczales, G. (2003). Back to the future: a retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd international conference on Aspect-oriented software development* (pp. 50–59). ACM.

Corden, M. J., & Kreitzer, D. (2014). *Consistency of Floating-Point Results using the Intel® Compiler or Why doesn't my application always give the same answer?*

Crimi, G., Mantovani, F., Pivanti, M., Schifano, S. F., & Tripiccione, R. (2013). Early experience on porting and running a Lattice Boltzmann code on the Xeon-Phi co-processor. In *Procedia Computer Science* (Vol. 18, pp. 551–560). http://doi.org/10.1016/j.procs.2013.05.219

Dagum, L., & Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *Computational Science \& Engineering, IEEE*, *5*(1), 46–55.

Dao, V. T., Maigne, L., Breton, V., Nguyen, H. Q., & Hill, D. R. C. (2014). Numerical Reproducibility, Portability And Performance Of Modern Pseudo Random Number Generators : Preliminary study for parallel stochastic simulations using hybrid Xeon Phi computing processors. In *European Simulation And Modelling Conference* (pp. 80–87). Porto.

De Bruijn, N. G. (1967). Additional comments on a problem in concurrent programming control. *Communications of the ACM*, *10*(3), 137–138.

Dedu, E., Vialle, S., & Timsit, C. (2000). Comparison of OpenMP and Classical Multi-Threading Parallelization for Regular and Irregular Algorithms. *Context*, *2*, 20.

Delisle, P., Krajecki, M., Gravel, M., & Gagné, C. (2001). Parallel implementation of an ant colony optimization metaheuristic with OpenMP. In *Proceedings of the 3rd European Workshop on OpenMP (EWOMP'01)*. Barcelona.

Dijkstra, E. W. (1965). Solution of a problem in concurrent programming control. *Communications of the ACM*, *8*(9), 569.

Dijkstra, E. W. (1968). The structure of THE multiprogramming system. *Communications of the ACM*, *11*(5), 341–346.

Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.-T., & Jalby, W. (2005). Maqao: Modular assembler quality analyzer and optimizer for itanium 2. In *The 4th Workshop on EPIC architectures and compiler technology, San Jose*.

Dokulil, J., Bajrovic, E., Benkner, S., Pllanaa, S., Sandrieser, M., & Bachmayer, B. (2013). High-level support for hybrid parallel execution of C++ applications targeting Intel® - Xeon Phi™ coprocessors. In *Procedia Computer Science* (Vol. 18, pp. 2508–2511). http://doi.org/10.1016/j.procs.2013.05.430

Dongarra, J. J., Bunch, J. R., Moler, C. B., & Stewart, G. W. (1979). *LINPACK users' guide*. Siam.

Drake, A. (2014). Command-line tools can be 235x faster than your Hadoop cluster. Retrieved March 6, 2015, from http://aadrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html

Dubach, C., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F., & Temam, O. (2007). Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th international conference on Computing frontiers* (pp. 131–142). ACM.

Eisenberg, M. A., & McGuire, M. R. (1972). Further comments on Dijkstra's concurrent programming control problem. *Communications of the ACM*, *15*(11), 1972.

El Bitar, Z., Lazaro, D., Coello, C., Breton, V., Hill, D. R. C., & Buvat, I. (2006). Fully 3D Monte Carlo image reconstruction in SPECT using functional regions. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, *569*(2), 399–403.

Eranian, S. (2006). Perfmon2: a flexible performance monitoring interface for Linux. In *Ottawa Linux Symposium* (pp. 269–288). Citeseer.

Estérie, P., Gaunard, M., Falcou, J., Lapresté, J.-T., & Rozoy, B. (2014). Boost. simd: generic programming for portable simdization. In *Proceedings of the 2014 Workshop on Workshop on programming models for SIMD/Vector processing* (pp. 1–8). ACM.

Evers, M., Chang, P.-Y., & Patt, Y. N. (1996). Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. *ACM SIGARCH Computer Architecture News*, *24*(2), 3–11.

Falcou, J., & Sérot, J. (2004). Eve, an object oriented simd library. In *Computational Science-ICCS 2004* (pp. 314–321). Springer.

Falcou, J., Sérot, J., Chateau, T., & Lapresté, J.-T. (2006). Quaff: efficient c++ design for parallel skeletons. *Parallel Computing*, *32*(7), 604–615.

Fehr, F. (2012). Density imaging of volcanos with atmospheric muons. *Journal of Physics: Conference Series*, *375*(5), 052019. http://doi.org/10.1088/1742-6596/375/1/052019

Fenlason, J., & Stallman, R. (1988). *GNU gprof*. Free Software Foundation, Inc.

Fleegal, E. (2004). *Microsoft Visual C++ Floating-Point Optimization*.

Frigo, M., Leiserson, C. E., Prokop, H., & Ramachandran, S. (1999). Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on* (pp. 285–297). IEEE.

Fujimoto, R. M. (2001). Parallel simulation: parallel and distributed simulation systems. In *Proceedings of the 33nd conference on Winter simulation* (pp. 147–157). IEEE Computer Society Press.

Fürlinger, K., & Gerndt, M. (2008). ompP: A profiling tool for OpenMP. In *OpenMP Shared Memory Parallel Programming* (pp. 15–23). Springer.

Garland, M., Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., & Volkov, V. (2008). Parallel Computing in CUDA. *IEEE Micro*, *28*(4), 13–27.

Gepner, P., Gamayunov, V., & Fraser, D. L. (2011). Early performance evaluation of AVX for HPC. In *Procedia Computer Science* (Vol. 4, pp. 452–460). Elsevier. http://doi.org/10.1016/j.procs.2011.04.047

Ghosh, S., Martonosi, M., & Malik, S. (2000). Automated cache optimizations using CME driven diagnosis. In *Proceedings of the 14th international conference on Supercomputing* (pp. 316–326). ACM.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, *23*(1), 5–48.

Gordon, R. (1998). *Essential JNI: Java Native Interface*. Prentice-Hall, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=SERIES9932.275573

Gourgand, M., & Hill, D. R. C. (1990). Petri Nets Modelling on Transputers. In *SCS European Simulation Symposium ESS 90* (pp. 143–148). Ghent.

Gradecki, J. D., & Lesiecki, N. (2003). *Mastering AspectJ: aspect-oriented programming in Java*. John Wiley & Sons.

Graham, S. L., Kessler, P. B., & Mckusick, M. K. (1982). Gprof: A call graph execution profiler. *ACM Sigplan Notices*, *17*(6), 120–126. Retrieved from http://dl.acm.org/citation.cfm?id=806987

Guo, P. J., & Engler, D. R. (2011). CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *USENIX Annual Technical Conference*.

Gustavson, F. G., Moreira, J. E., & Enenkel, R. F. (1999). The fused multiply-add instruction leads to algorithms for extended-precision floating point: applications to java and high-performance

computing. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (p. 4). IBM Press.

Hall, R. J. (2002). CPPROFJ: Aspect-capable call path profiling of multi-threaded Java applications. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on* (pp. 107–116). IEEE.

Haramoto, H., Matsumoto, M., & L'Ecuyer, P. (2008). A fast jump ahead algorithm for linear recurrences in a polynomial space. In *Sequences and Their …* (pp. 290–298). Springer. Retrieved from http://link.springer.com/chapter/10.1007/978-3-540-85912-3_26

Hellekalek, P. (1998). Don't trust parallel Monte Carlo! *ACM SIGSIM Simulation Digest*, *28*(1), 82–89.

Henderson, P. (1980). *Functional programming: application and implementation*. Prentice-Hall, Inc.

Hill, D. R. C. (2015). Parallel Random Numbers, Science and reproducibility. *IEEE Computing in Science and Engineering*, *17*(4), 66–71.

Hill, D. R. C., Mazel, C., Passerat-Palmbach, J., & Traore, M. K. (2013). Distribution of random streams for simulation practitioners. *Concurrency and Computation: Practice and Experience*, *25*(10), 1427–1442. http://doi.org/10.1002/cpe.2942

Hulth, P. (1996). The AMANDA experiment. *arXiv Preprint Astro-ph/9612068*. Retrieved from http://arxiv.org/abs/astro-ph/9612068

IEEE. (2008). IEEE Standard for Floating-Point Arithmetic. In *IEEE Std 754-2008* (pp. 1 – 70). IEEE.

Innocenti, E. (2004). *Contribution à la modélisation de systèmes spatiaux implémentés dans des environnements parallèles et distribués*. Université Pasquale Paoli.

Innocenti, E., Silvani, X., Muzy, A., & Hill, D. R. C. (2009). A software framework for fine grain parallelization of cellular models with OpenMP: Application to fire spread. *Environmental Modelling & Software*, *24*(7), 819–831.

Intel. (2010). *Quick-Reference Guide to Optimization with Intel® Compilers version 12*. Retrieved from https://software.intel.com/sites/default/files/compiler_qrg12.pdf

Intel. (2014). *Differences in Floating-Point Arithmetic Between Intel® Xeon® Processors and the Intel® Xeon Phi Coprocessor*.

Kernighan, B. W., & Plauger, P. J. (1978). *The elements of programming style*. McGraw-Hill.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-oriented programming. *ECOOP'97 — Object-Oriented Programming*, *1241*, 220–242. Retrieved from http://link.springer.com/chapter/10.1007/BFb0053381

Kirschenmann, W., Plagne, L., & Vialle, S. (2012). Multi-target vectorization with MTPS C++ generic library. In *Applied Parallel and Scientific Computing* (pp. 336–346). Springer.

Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., & Malony, A. (2012). Score-p: a joint performance measurement run-time

infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011* (pp. 79–91). Springer.

Knuth, D. (1966). Additional comments on a problem in concurrent programming control. *Communications of the ACM*, *9*(5), 321–322.

Knuth, D. (1969). Semi-numerical algorithms. In *The Art of Computer Programming* (Vol. 2). Addison-Wesley.

L'Ecuyer, P. (1999). Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, *47*(1), 159–164.

L'Ecuyer, P. (2010). Pseudorandom number generator. *Encyclopedia of Quantitative Finance*, *Simulation*.

L'Ecuyer, P., Oreshkin, B., & Simard, R. (2014). Random Numbers for Parallel Computers: Requirements and Methods. *To Appear*.

L'Ecuyer, P., & Simard, R. (2007). TestU01: AC library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, *33*(4), 22. Retrieved from http://dl.acm.org/citation.cfm?id=1268777

L'Ecuyer, P., Simard, R., Chen, E. J., & Kelton, W. D. (2002). An object-oriented random-number package with many long streams and substreams. *Operations Research*, *50*(6), 1073–1075.

Lamport, L. (1974). A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, *17*(8), 453–455.

Lange, B., & Fortin, P. (2014). *Parallel dual tree traversal on multi-core and many-core architectures for astrophysical N-body simulations*. Retrieved from http://hal.upmc.fr/hal-00947130v1

Lattner, C. (2008). LLVM and Clang: Next generation compiler technology. In *The BSD Conference* (pp. 1–2).

Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (pp. 75–86). IEEE.

Lazaro, D., El Bitar, Z., Breton, V., Hill, D. R. C., & Buvat, I. (2005). Fully 3D Monte Carlo reconstruction in SPECT: a feasibility study. *Physics in Medicine and Biology*, *50*(16), 3739–3754.

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., & Dubey, P. (2010). Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, *38*(3), 451–460.

Leng, T., Ali, R., Hsieh, J., Mashayekhi, V., & Rooholamini, R. (2002). An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*. Retrieved from http://www.democritos.it/activities/IT-MC/cluster_revolution_2002/PDF/11-Leng_T.pdf

Levinthal, D. (2009). *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors*. Retrieved from https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

Levon, J., & Elie, P. (2004). Oprofile: A system profiler for linux.

Li, C., Ding, C., & Shen, K. (2007). Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science* (p. 4). ACM.

Li, S. (2013). *Case Study: Achieving High Performance on Monte Carlo European Option Using Stepwise Optimization Framework*. Retrieved from https://software.intel.com/en-us/articles/case-study-achieving-high-performance-on-monte-carlo-european-option-using-stepwise

Li, T., Baumberger, D., Koufaty, D. A., & Hahn, S. (2007). Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing* (p. 53). ACM Press. Retrieved from http://portal.acm.org/citation.cfm?doid=1362622.1362694

Liao, C., Yan, Y., de Supinski, B. R., Quinlan, D. J., & Chapman, B. (2013). Early experiences with the OpenMP accelerator model. In *OpenMP in the Era of Low Power Devices and Accelerators* (pp. 84–98). Springer.

Liu, T., Xu, X. G., & Carothers, C. D. (2014). Comparison of two accelerators for Monte Carlo radiation transport calculations, Nvidia Tesla M2090 GPU and Intel Xeon Phi 5110p coprocessor: A case study for X-ray CT imaging dose calculation. *Annals of Nuclear Energy*. http://doi.org/10.1016/j.anucene.2014.08.061

Lohmann, W., Kopp, R., & Voss, R. (1985). Energy Loss of Muons in the Energy Range 1-10000 GeV. In *CERN Report 85-03*.

Lönnblad, L. (1994). CLHEP—a project for designing a C++ class library for high energy physics. *Computer Physics Communications*, *84*(1), 307–316.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., & Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, *40*(6), 190–200.

Lüscher, M. (1994). A portable high-quality random number generator for lattice field theory simulations. *Computer Physics Communications*, *79*(1), 100–110. Retrieved from http://www.sciencedirect.com/science/article/pii/0010465594902321

Luszczek, P. R., Bailey, D. H., Dongarra, J. J., Kepner, J., Lucas, R. F., Rabenseifner, R., & Takahashi, D. (2006). The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. Citeseer.

Lyakh, D. I. (2015). An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU. *Computer Physics Communications*, *189*, 84–91. http://doi.org/10.1016/j.cpc.2014.12.013

Malony, A. D., Mellor-Crummey, J., & Shende, S. S. (2011). Measurement and Analysis of Parallel Program Performance Using TAU and HPCToolkit. In *Performance Tuning of Scientific Applications* (pp. 49–86).

Marr, D. T. (2002). Hyper-threading technology architecture and microarchitecture: a hyperhtext history. *Intel Technology Journal*.

Marsaglia, G. (1996). DIEHARD: a battery of tests of randomness. Retrieved from http://www.stat.fsu.edu/pub/diehard/

Martin, K., Hoffman, B., Cedilnik, A., King, B., & Nuendorf, A. (2010). *Mastering CMake: A cross-platform build system*. Kitware Incorporated.

Martin-Haugh, S. (2013). *Performance and development plans for the Inner Detector trigger algorithms at ATLAS*. Retrieved from http://cds.cern.ch/record/1623129/files/ATL-DAQ-PROC-2013-031.pdf

Mascagni, M., & Srinivasan, A. (2000). Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software (TOMS)*, *26*(3), 436–461.

Matsumoto, M., & Nishimura, T. (1998a). Dynamic creation of pseudorandom number generators. *Monte Carlo and Quasi-Monte Carlo Methods Conference*, 56–69. Retrieved from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dgene.pdf

Matsumoto, M., & Nishimura, T. (1998b). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, *8*(1), 3–30. Retrieved from http://dl.acm.org/citation.cfm?id=272995

Mazouz, A., Touati, S. A. A., & Barthou, D. (2010). Study of variations of native program execution times on multi-core architectures. In *CISIS 2010 - The 4th International Conference on Complex, Intelligent and Software Intensive Systems* (pp. 919–924). Cracovie: IEEE. http://doi.org/10.1109/CISIS.2010.96

Mazouz, A., Touati, S. A. A., & Barthou, D. (2011). Analysing the Variability of OpenMP Programs Performances on Multicore Architectures. In *Fourth Workshop on Programmability Issues for Heterogeneous Multicore* (p. 14). Heraklion.

McCalpin, J. D. (1995). *STREAM: Sustainable memory bandwidth in high performance computers*.

McCalpin, J. D. (1999). A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, 19–25.

McDougall, R., Mauro, J., & Gregg, B. (2006). *Solaris (TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Prentice Hall PTR.

McKinley, K. S., Carr, S., & Tseng, C.-W. (1996). Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *18*(4), 424–453.

Mohr, B., Malony, A. D., Shende, S., & Wolf, F. (2002). Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, *23*(1), 105–128.

Mucci, P. J., Browne, S., Deane, C., & Ho, G. (1999). PAPI: A portable interface to hardware performance counters. In *Proc. Department of Defense HPCMP Users Group Conference*. Retrieved from http://web.eecs.utk.edu/~mucci/latest/pubs/dodugc99-papi.pdf

Murano, K., Shimobaba, T., Sugiyama, A., Takada, N., Kakue, T., Oikawa, M., & Ito, T. (2014). Fast computation of computer-generated hologram using Xeon Phi coprocessor. *Computer Physics Communications*, *185*(10), 2742–2757. http://doi.org/10.1016/j.cpc.2014.06.010

Nagel, W. E., Arnold, A., Weber, M., Hoppe, H.-C., & Solchenbach, K. (1996). VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, *12*(1), 69–80.

Nethercote, N., & Seward, J. (2003). Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, *89*(2), 44–66. Retrieved from http://www.sciencedirect.com/science/article/pii/S1571066104810429

Nowak, A. (2010). *Strategies employed for LHC software performance studies*. Retrieved from http://cds.cern.ch/record/1325106/files/01-Strategies employed for LHC software performance studies FINAL.pdf

OpenACC Working Group. (2011). *The OpenACC Application Programming Interface*.

OpenHMPP Consortium. (2011). OpenHMPP Concepts and Directives.

Pan, S.-T., So, K., & Rahmeh, J. T. (1992). Improving the accuracy of dynamic branch prediction using branch correlation. *ACM Sigplan Notices*, *27*(9), 76–84.

Pan, Z., & Eigenmann, R. (2006). Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on* (pp. 319–332). IEEE.

Pearce, D. J., Webster, M., Berry, R., & Kelly, P. H. J. (2007). Profiling with AspectJ. *Software: Practice and Experience*, *37*(7), 747–777.

Peleg, A., & Weiser, U. (1996). MMX technology extension to the Intel architecture. *Micro, IEEE*, *16*(4), 42–50.

Pettis, K., & Hansen, R. C. (1990). Profile guided code positioning. *ACM SIGPLAN Notices*, *25*(6), 16–27.

Pfister, G. F. (2001). An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, *42*, 617–632.

Pharr, M., & Mark, W. R. (2012). ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012* (pp. 1–13).

Pheatt, C. (2008). Intel threading building blocks. *Journal of Computing Sciences in Colleges*, *23*(4), 298–298.

Pietrek, A., Bouchez, F., & de Dinechin, B. D. (2011). Tirex: A textual target-level intermediate representation for compiler exchange. In *Workshop on Intermediate Representations* (pp. 13–20). Chamonix.

Plauger, P. J., Lee, M., Musser, D., & Stepanov, A. A. (2000). *C++ Standard Template Library*. Prentice Hall PTR. Retrieved from http://dl.acm.org/citation.cfm?id=557814

Press, T. (1994). Taligent's Guide to Designing Programs. *Reading, Massachusetts*. Retrieved from http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Taligent's+Guide+to+Designing+Programs#0

Radon, J. (1917). Über die Bestimmung von Funktionen durch ihre Integralwerte längs gewisser Mannigfaltigkeiten. *Berichte Über Die Verhandlungen Der Königlich-Sächsischen Akademie Der Wissenschaften Zu Leipzig, Mathematisch-Physische Klasse*, *69*, 262–277.

Radon, J. (1986). On the determination of functions from their integral values along certain manifolds. *IEEE Transactions on Medical Imaging*, *5*(4), 170–176. http://doi.org/10.1109/TMI.1986.4307775

Rajasekaran, S., & Reif, J. (2007). *Handbook of parallel computing: models, algorithms and applications*. CRC Press.

Reddi, V. J., Settle, A., Connors, D. A., & Cohn, R. S. (2004). PIN: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture* (p. 22). ACM.

Rentsch, T. (1982). Object oriented programming. *ACM SIGPLAN Notices*. http://doi.org/10.1145/947955.947961

Reuillon, R., Hill, D. R. C., El Bitar, Z., & Breton, V. (2008). Rigorous distribution of stochastic simulations using the DistMe toolkit. *IEEE Transactions on Nuclear Science*, *55*(1), 595–603.

Richardson, S., & Ganapathi, M. (1989). Interprocedural optimization: Experimental results. *Software: Practice and Experience*, *19*(2), 149–169. http://doi.org/10.1002/spe.4380190205

Rivera, G., & Tseng, C.-W. (1998). Data transformations for eliminating conflict misses. *ACM SIGPLAN Notices*, *33*(5), 38–49.

Rosenquist, T. (2012). *Run-to-run Numerical Reproducibility with the Intel® Math Kernel Library and Intel® Composer XE 2013*.

Sabahi, M. (2012). *A Guide to Auto-vectorization with Intel® C++ Compilers*. Retrieved from https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers

Salmon, J. K., Moraes, M. A., Dror, R. O., & Shaw, D. E. (2011). Parallel random numbers: as easy as 1, 2, 3. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for* (pp. 1–12). IEEE.

Schwab, M., Karrenbach, M., & Claerbout, J. (2000). Making scientific computations reproducible. *Computing in Science & Engineering*, *2*(6), 61–67.

Schweitzer, P., Mazel, C., Cârloganu, C., & Hill, D. R. C. (2015). Performance Analysis with a Memory-Bound Monte Carlo Simulation on Xeon Phi. In *Proceedings of the 2015 International Conference on High Performance Computing & Simulation* (pp. 444–452).

Schweitzer, P., Mazel, C., Fehr, F., Cârloganu, C., & Hill, D. R. C. (2013). Proper parallel Monte Carlo for computed tomography of volcanoes. In *Proceedings of the 2013 International Conference on High Performance Computing & Simulation* (pp. 519–526).

Sfiligoi, I., Quinn, G., Green, C., & Thain, G. (2008). Pilot job accounting and auditing in Open Science Grid. In *Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing* (pp. 112–117). IEEE.

Shende, S., Malony, A. D., Cuny, J., Beckman, P., Karmesin, S., & Lindlan, K. (1998). Portable profiling and tracing for parallel, scientific applications using C++. In ACM (Ed.), *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (pp. 134–145).

Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (pp. 1–10). IEEE.

Smith, J. E. (1981). A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture* (pp. 135–148). IEEE Computer Society Press.

Snir, M., Otto, S. W., Walker, D. W., Dongarra, J., & Huss-Lederman, S. (1995). *MPI: the complete reference*. MIT Press. Retrieved from http://dl.acm.org/citation.cfm?id=546703

Spinczyk, O., Gal, A., & Schröder-Preikschat, W. (2002). AspectC++: an aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications* (pp. 53–60). Retrieved from http://dl.acm.org/citation.cfm?id=564100

Stone, J. E., Gohara, D., & Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, *12*(1-3), 66–73.

Stravers, P., & Van De, W. J. (2004). Translation Lookaside Buffer. US.

Tao, J. (2010). Comprehensive cache performance tuning with a toolset. *Future Generation Computer Systems*, *26*, 167–174. http://doi.org/10.1016/j.future.2009.07.010

Tomar, S. (2006). Converting video formats with FFmpeg. *Linux Journal*, *2006*(146).

Tullsen, D. M., Eggers, S. J., & Levy, H. M. (1995). Simultaneous multithreading: Maximizing on-chip parallelism. *ACM SIGARCH Computer Architecture News*, *23*(2), 392–403.

Urbatsch, T. J., & Evans, T. M. (1999). Reproducibility in parallel Monte Carlo codes. *Los Alamos National Laboratory Memorandum*, *12*.

Valensi, C., & Barthou, D. (2009). MADRAS: Multi-Architecture Disassembler and Reassembler. Retrieved from http://maqao.prism.uvsq.fr/wiki/wiki/MadrasDownload

Valgrind-project. (2007). *Helgrind: a data-race compressor*.

Van Rossum, G. (2010). *The Python Language Reference*. (F. L. Drake, Ed.). Python Software Foundation.

Venetis, I. E., Goumas, G., Geveler, M., & Ribbrock, D. (2014). Porting FEASTFLOW to the Intel Xeon Phi: Lessons Learned. *Partnership for Advanced Computing in Europe (PRACE)*, *139*.

Vetter, J., & Chambreau, C. (2005). *mpip: Lightweight, scalable mpi profiling*.

Vladimirov, A., & Karpusenko, V. (2013). *Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors*. Colfax.

Wang, Z., Sha, E. H.-M., & Hu, X. S. (2001). Combined partitioning and data padding for scheduling multiple loop nests. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems* (pp. 67–75). ACM.

Weidendorfer, J., Kowarschik, M., & Trinitis, C. (2004). A tool suite for simulation based analysis of memory access behavior. *Computational Science-ICCS 2004*, 440–447. Retrieved from http://link.springer.com/chapter/10.1007/978-3-540-24688-6_58

Wolf, F., Wylie, B. J., Abraham, E., Becker, D., Frings, W., Fürlinger, K., Geimer, M., Hermanns, M.-A., Mohr, B., Moore, S., & Szebenyi, Z. (2008). Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In *Tools for High Performance Computing* (pp. 157–167). Stuttgart: Springer.

Wolf, J. H. (1999). Programming Methods for the Pentium III Processor's Streaming SIMD Extensions Using the VTune Performance Enhancement Environment. *Intel Technology Journal*.

Wulf, W. A., & McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, *23*(1), 20–24.

Xu, R., Chandrasekaran, S., & Chapman, B. (2013). Exploring programming multi-GPUS using OpenMP and OpenACC-based hybrid model. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* (pp. 1169–1176). IEEE.

Yeh, T.-Y., & Patt, Y. N. (1992). Alternative implementations of two-level adaptive branch prediction. *ACM SIGARCH Computer Architecture News*, *20*(2), 124–134.

Young, C., & Smith, M. D. (1994). Improving the accuracy of static branch prediction using branch correlation. *ACM Sigplan Notices*, *29*(11), 232–241.

Yu, Y., Beyls, K., & D'Hollander, E. H. (2001). Visualizing the impact of the cache on program execution. In *Information Visualisation, 2001. Proceedings. Fifth International Conference on* (pp. 336–341). IEEE.

Zaki, O., Lusk, E., Gropp, W., & Swider, D. (1999). Toward scalable performance visualization with Jumpshot. *International Journal of High Performance Computing Applications*, *13*(3), 277–288.